# *Algoritmi e Strutture Dati*

## Alberi Binari di Ricerca

# *Alberi binari di ricerca*

## Motivazioni

- **gestione e ricerche in grosse quantità di dati**
- ***liste*, *array* e *alberi non* sono *adeguati* perché inefficienti in tempo $O(n)$ o in spazio**

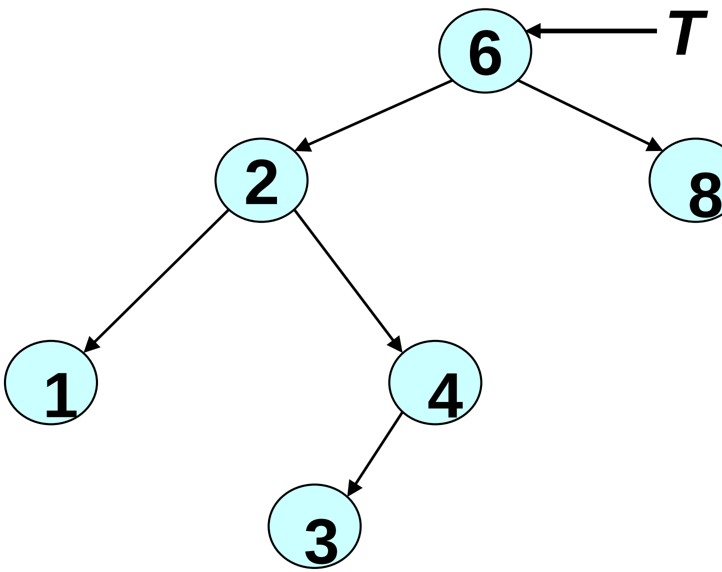
## Esempi:

- **Mantenimento di archivi (*DataBase*)**
- **In generale, mantenimento e gestione di corpi di *dati* su cui si effettuano *molte ricerche*, eventualmente alternate a operazioni di inserimento e cancellazione.**

# *Alberi binari di ricerca*

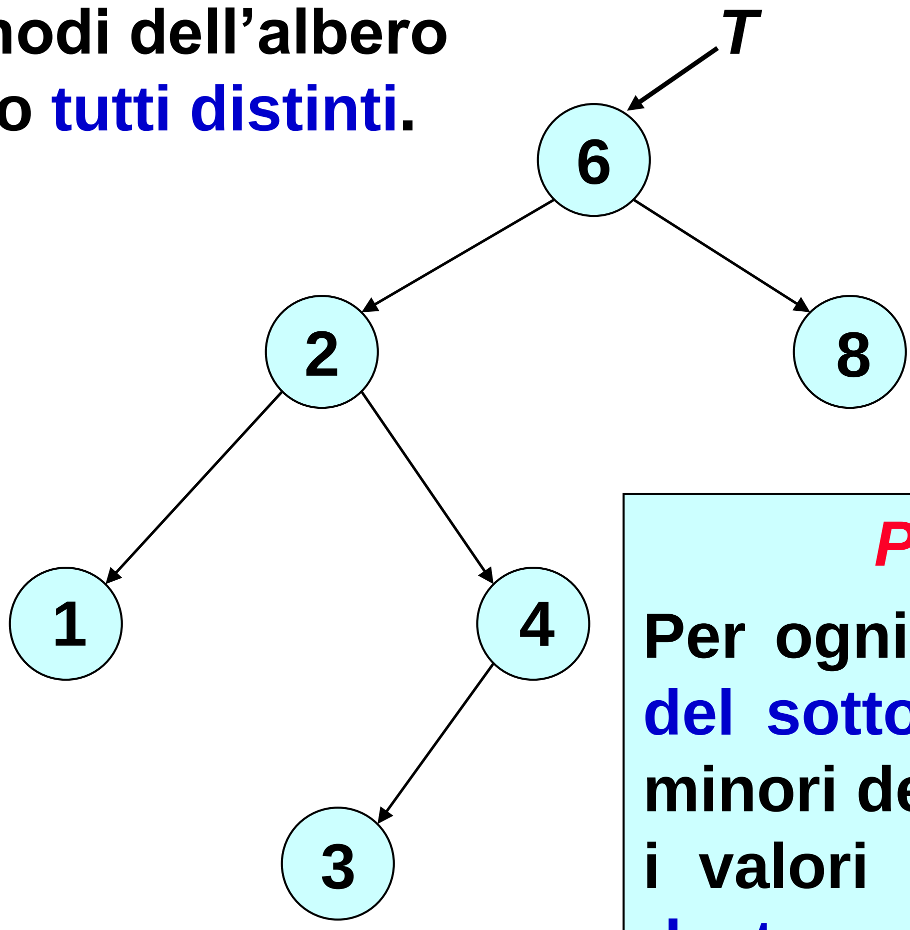
***Definizione*: Un *albero binario di ricerca* è un albero binario che soddisfa la seguente proprietà:**

> **se *X* è un nodo e *Y* è qualsiasi un nodo nel sottoalbero sinistro di *X*, allora *Y->key* ≤ *X->key*;**
>
> **inoltre, se *Y* è qualsiasi un nodo nel sottoablero destro di *X* allora *Y->key* ≥ *X->key***



# *Alberi binari di ricerca*

**Assumiamo che i valori nei nodi dell'albero siano tutti distinti.**



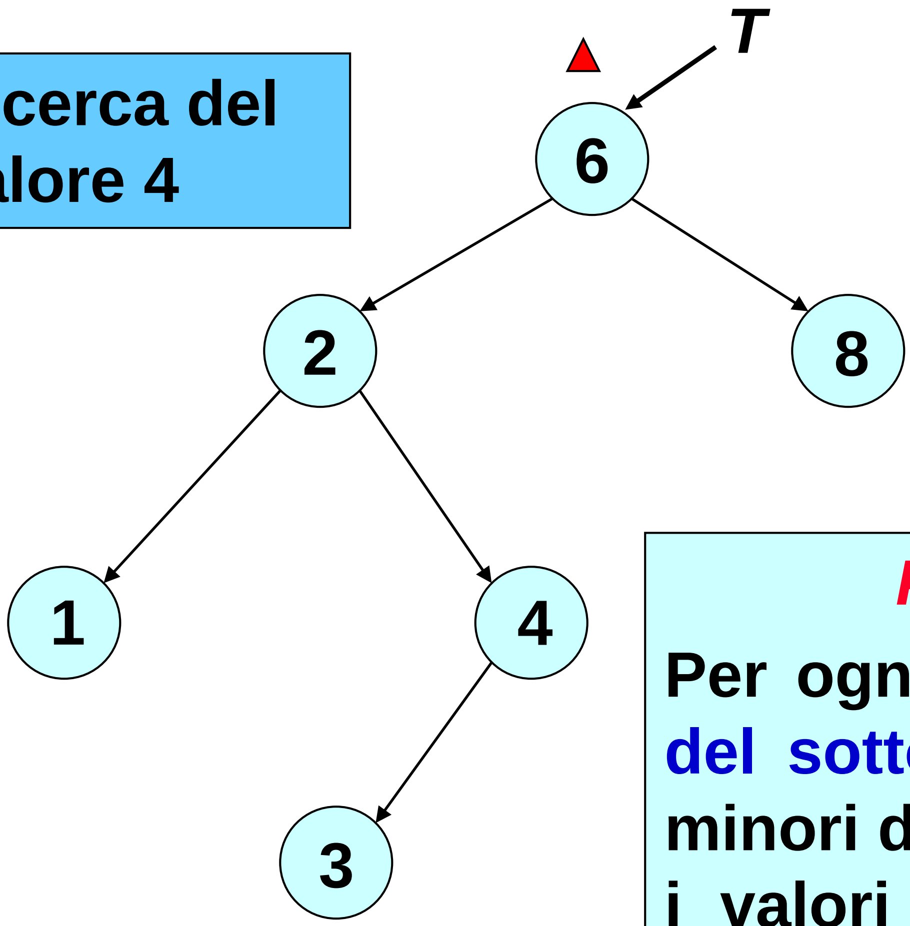

**Assumiamo che i valori nei nodi (le chiavi) pos-sano essere ordinati.**

***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*

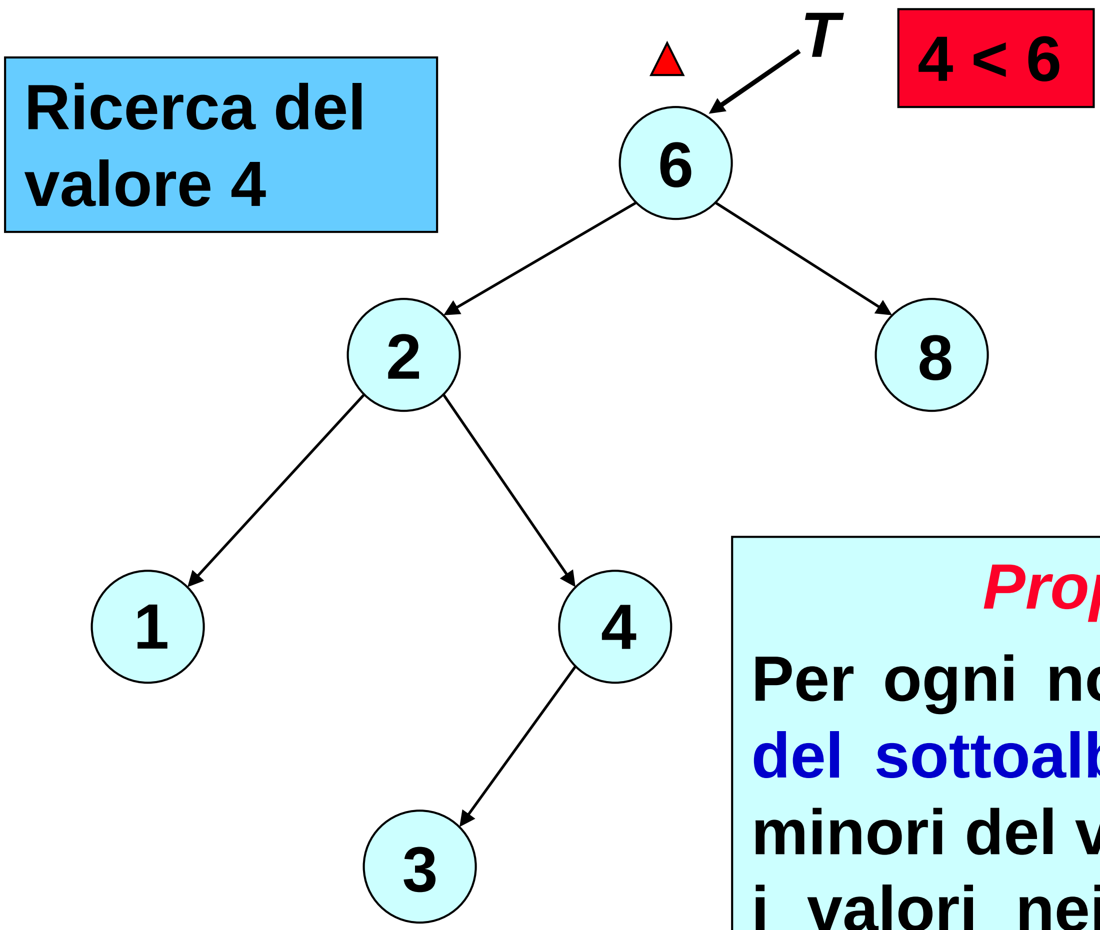
**Ricerca del valore 4**




***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*



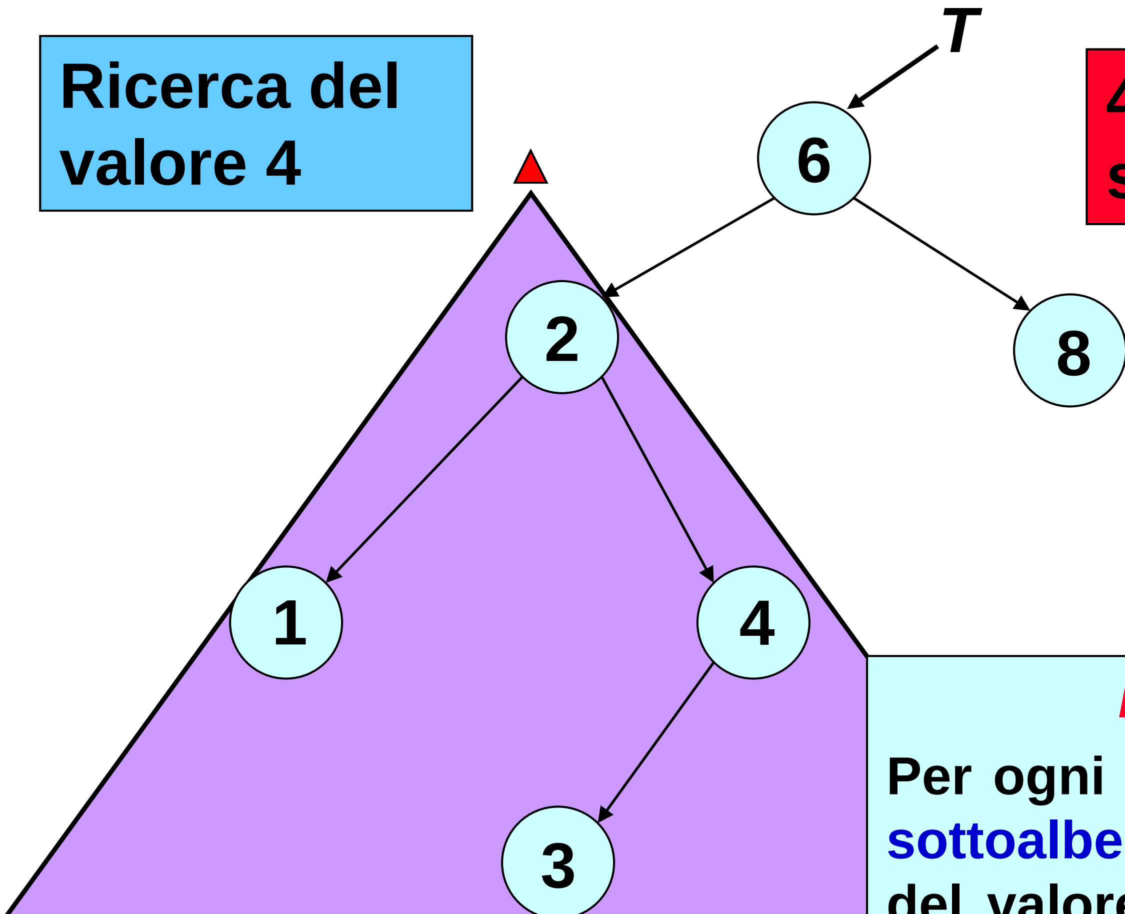

***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*

**Ricerca del valore 4**

**4 sta nel sottoalbero sinistro di 6**



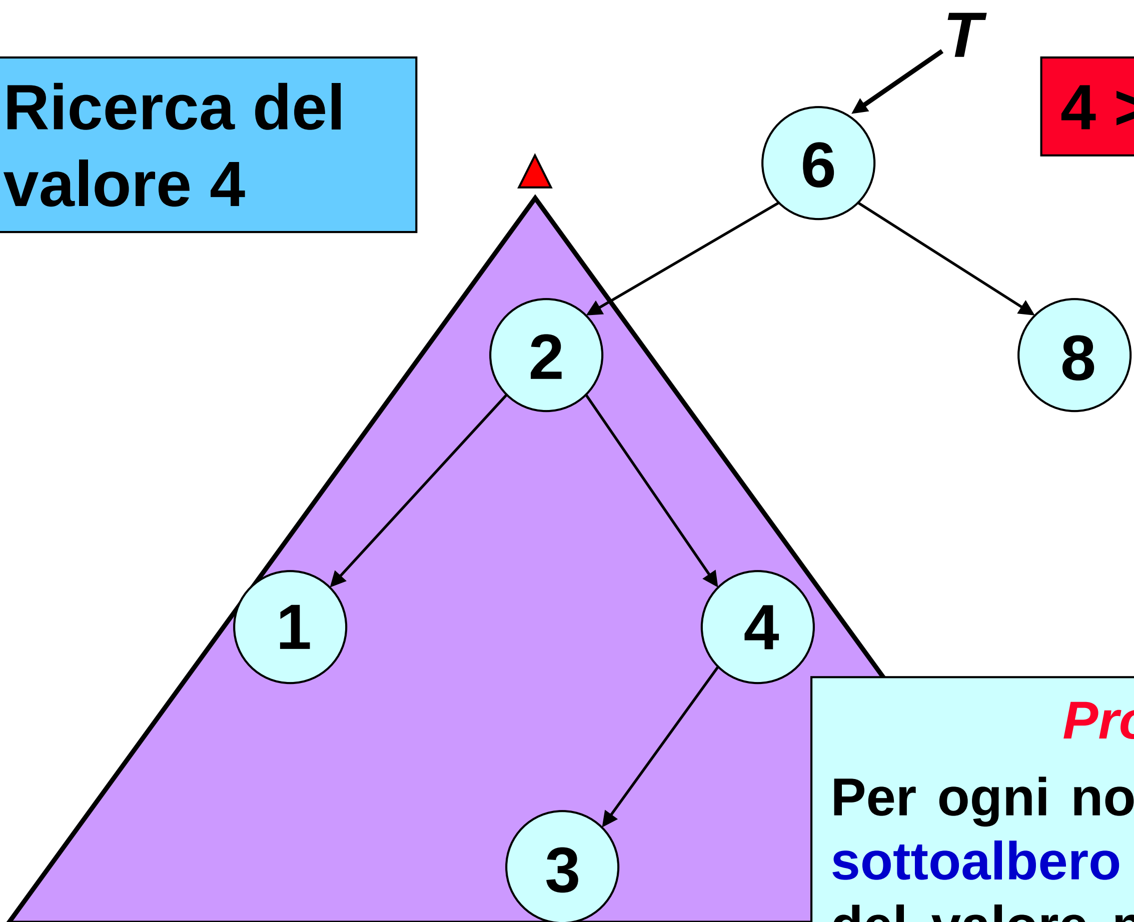

***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*

**Ricerca del valore 4**

**4 > 2**

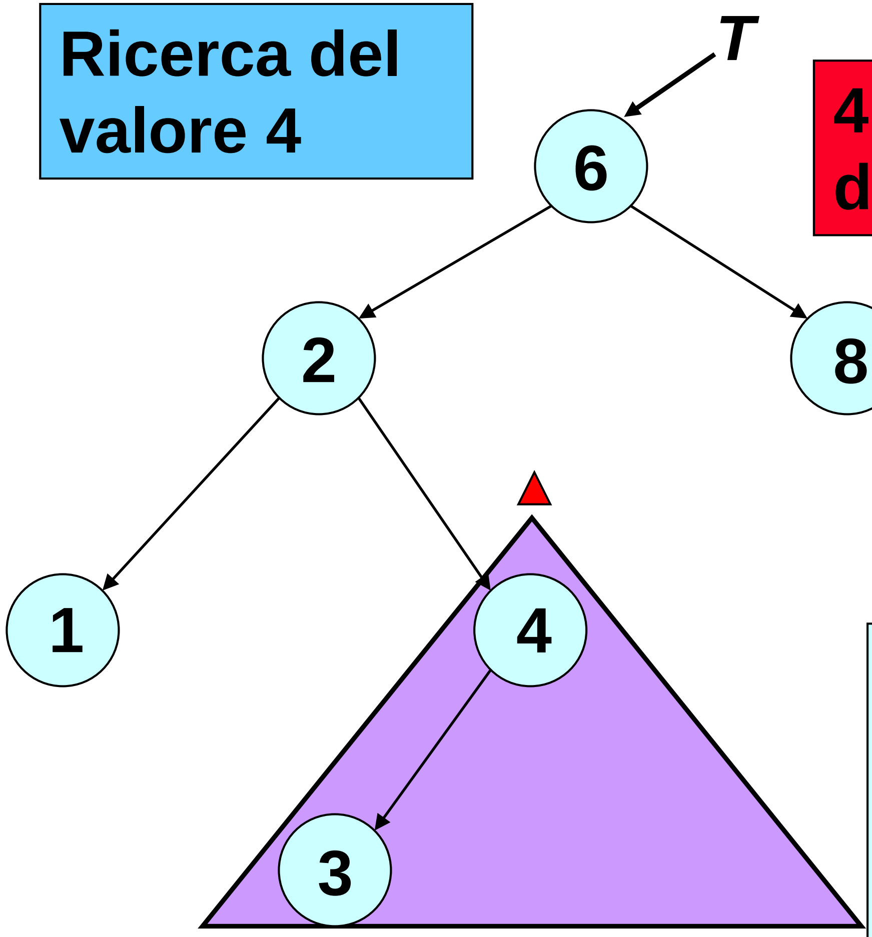



***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*

**Ricerca del valore 4**



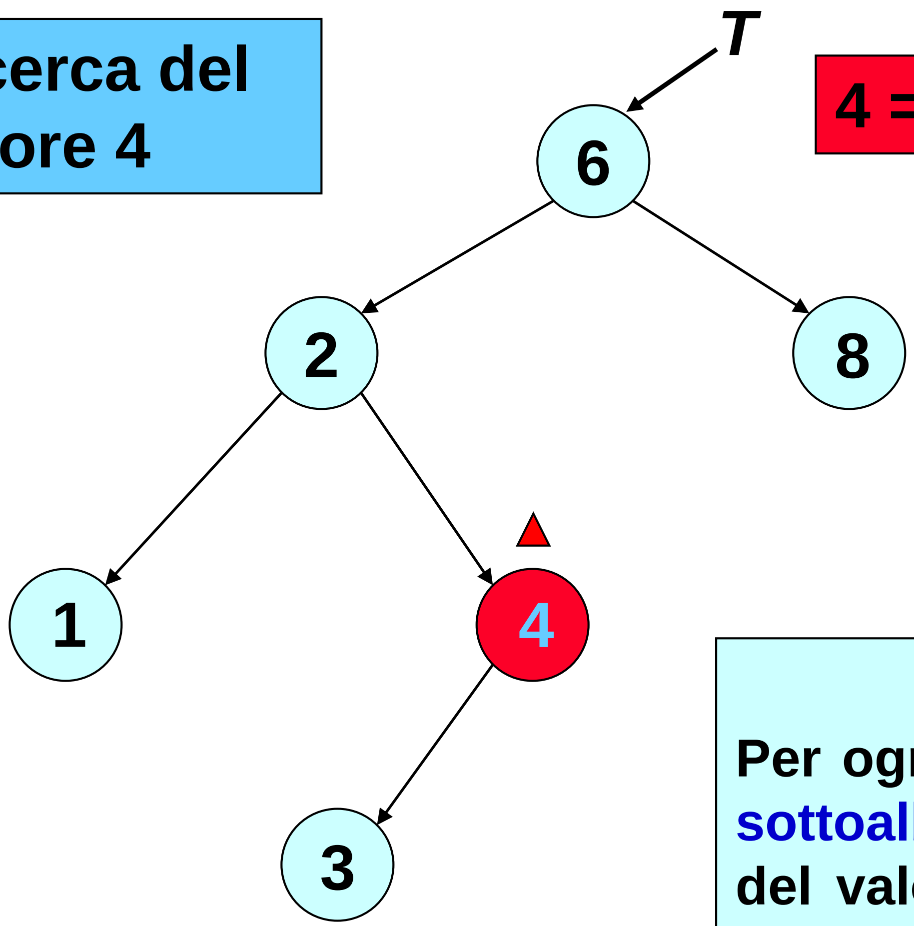

**4 sta nel sottoalbero destro di 2**

***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca: esempio*

**Ricerca del valore 4**




**4 = 4 : *Trovato*!**

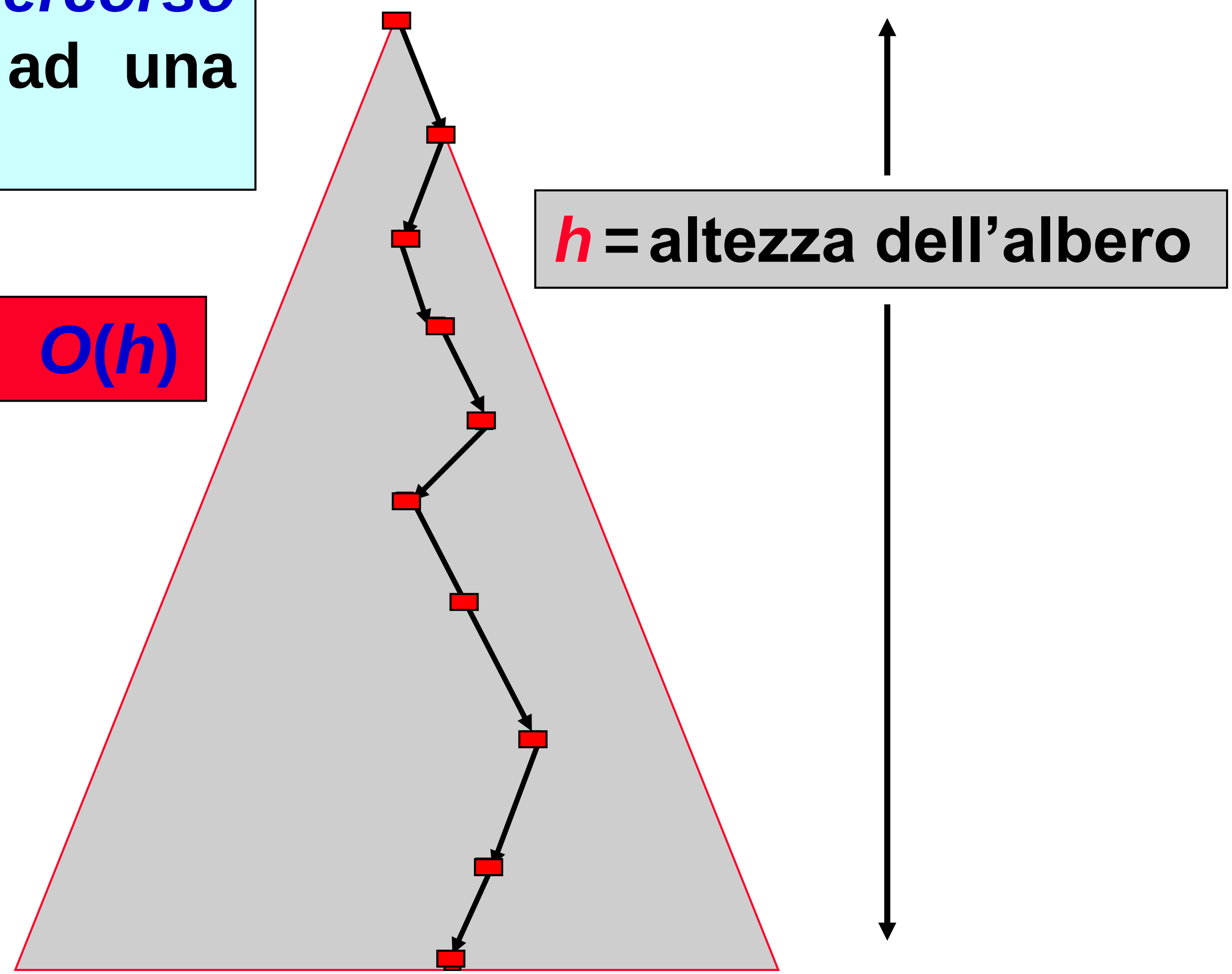
***Proprietà* degli *ABR***

**Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X**

# *Alberi binari di ricerca*

**In generale, la *ricerca* è confinata ai *nodi* posizionati *lungo un singolo percorso* (path) dalla radice ad una foglia**

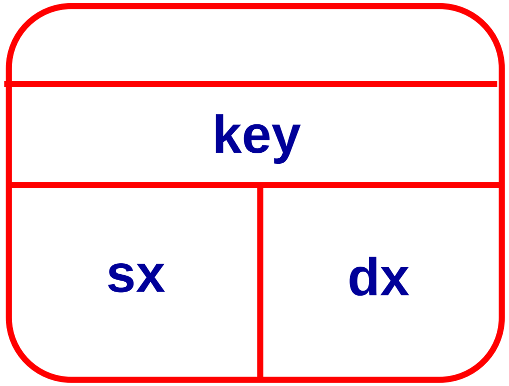
**Tempo di ricerca = *O*(*h*)**



# ADT albero binario di ricerca: tipo di dato

- *È una specializzazione dell'ADT albero binario*
- *Gli elementi statici sono essenzialmente gli stessi, l'unica differenza è che si assume che i dati contenuti (le chiavi) siano ordinabili secondo qualche relazione d'ordine.*

```
typedef *nodo ARB;
struct {
        elemento key;
        ARB sx, dx;
      } nodo;
```



# ADT albero binario di ricerca: funzioni

- **Selettori:**
  - *root(T)*
  - *dx(T)*
  - *sx(T)*
  - *key(T)*
- **Costruttori/Distruttori:**
  - *crea_albero()*
  - *ARB_inserisci(T,x)*
  - *ARB_cancella (T,x)*
- **Proprietà:**
  - *vuoto(T)* = return (T=Nil)

Ritorna il valore del test di uguaglianza

- **Operazioni di Ricerca**
  - *ARB_ricerca(T,k)*
  - *ARB_minimo(T)*
  - *ARB_massimo(T)*
  - *ARB_successore(T,x)*
  - *ARB_predecessore(T,x)*

# *Ricerca in Alberi binari di ricerca*

```
ARB_ricerca(T,k)
    IF T ≠ NIL THEN
         IF k ≠ T->Key THEN
             IF k < T->Key THEN
                return ARB_ricerca(T->sx,k)
             ELSE
                return ARB_ricerca(T->dx,k)
    return T
```

***NOTA:** Questo algoritmo cerca il nodo con chiave k nell'albero T e ne ritorna il puntatore. Ritorna* **NIL** *nel caso non esista alcun nodo con chiave k.*

# *Ricerca in Alberi binari di ricerca*

```
ARB_ricerca'(T,k)
   IF T = NIL OR k = T->Key THEN
        return T
   ELSE IF k < T->Key THEN
            return ARB_ricerca'(T->sx,k)
        ELSE
            return ARB_ricerca'(T->dx,k)
```

***NOTA:** Variante sintattica del precedente algoritmo!*

# Ricerca in Alberi binari di ricerca

```
ARB_ricerca(T,k)
   cur = T
   while cur ≠ NIL AND k ≠ cur->Key THEN
        IF k < T->Key THEN
            cur = cur->sx
        ELSE
            cur = cur->dx
   return cur
```

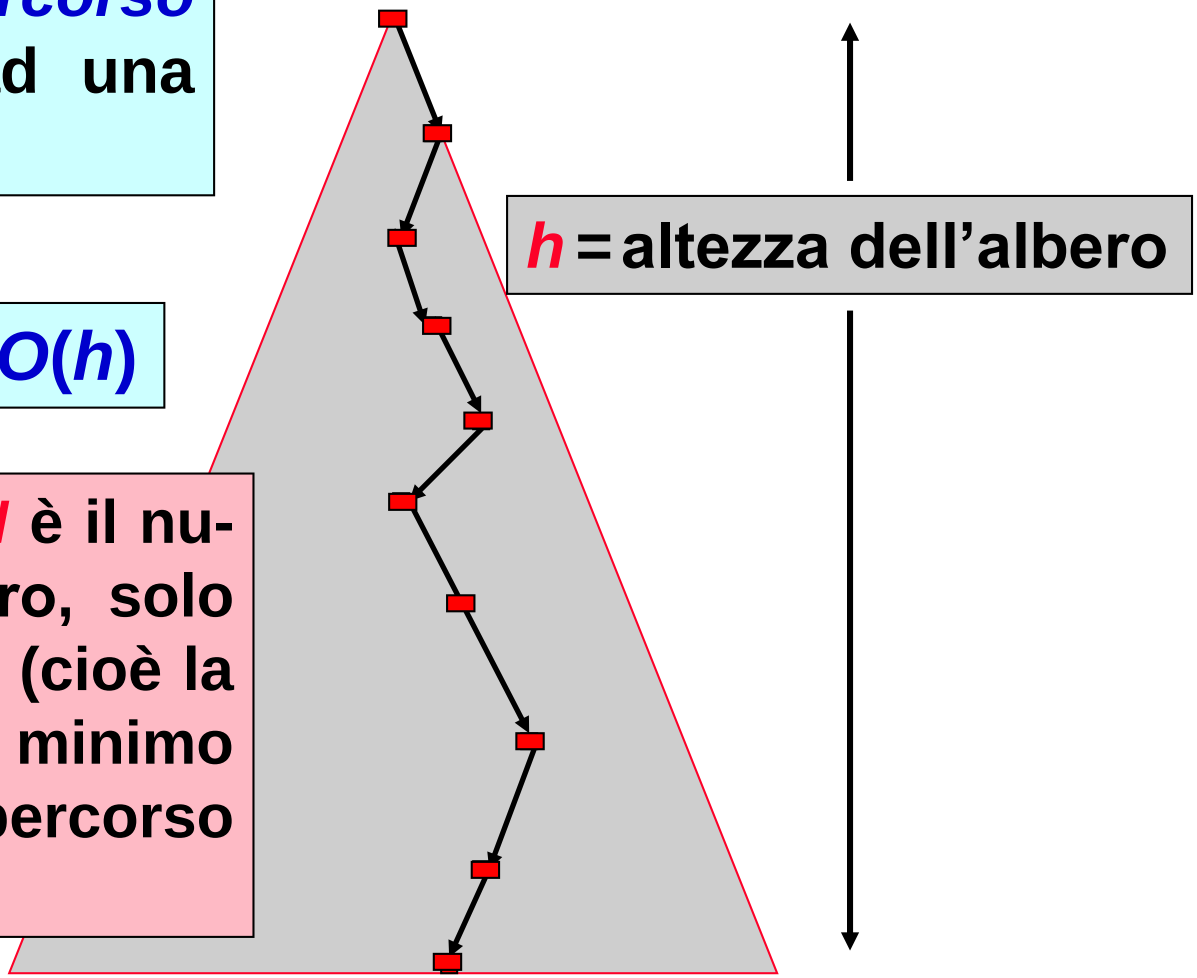
***NOTA: Versione iterativa!***

# *Ricerca in Alberi binari di ricerca*

**In generale, la *ricerca* è confinata ai *nodi* posizionati *lungo un singolo percorso* (*path*) dalla radice ad una foglia**

**Tempo di ricerca = $O(h)$**

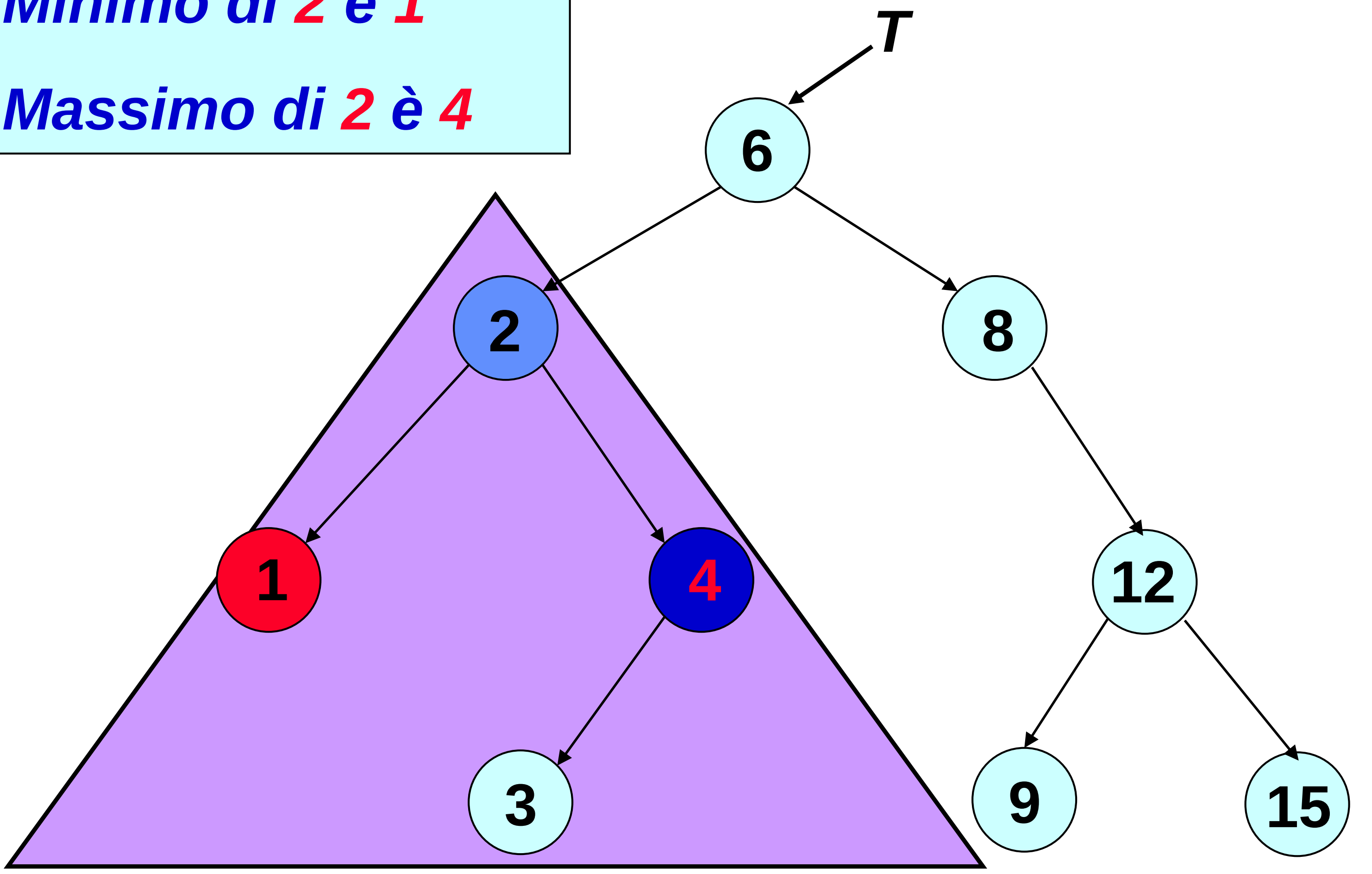
**$O(h)$ = $O(\log N)$, dove $N$ è il numero di nodi nell'albero, solo se l'albero è *balanciato* (cioè la lunghezza del percorso minimo è vicino a quella del percorso massimo).**



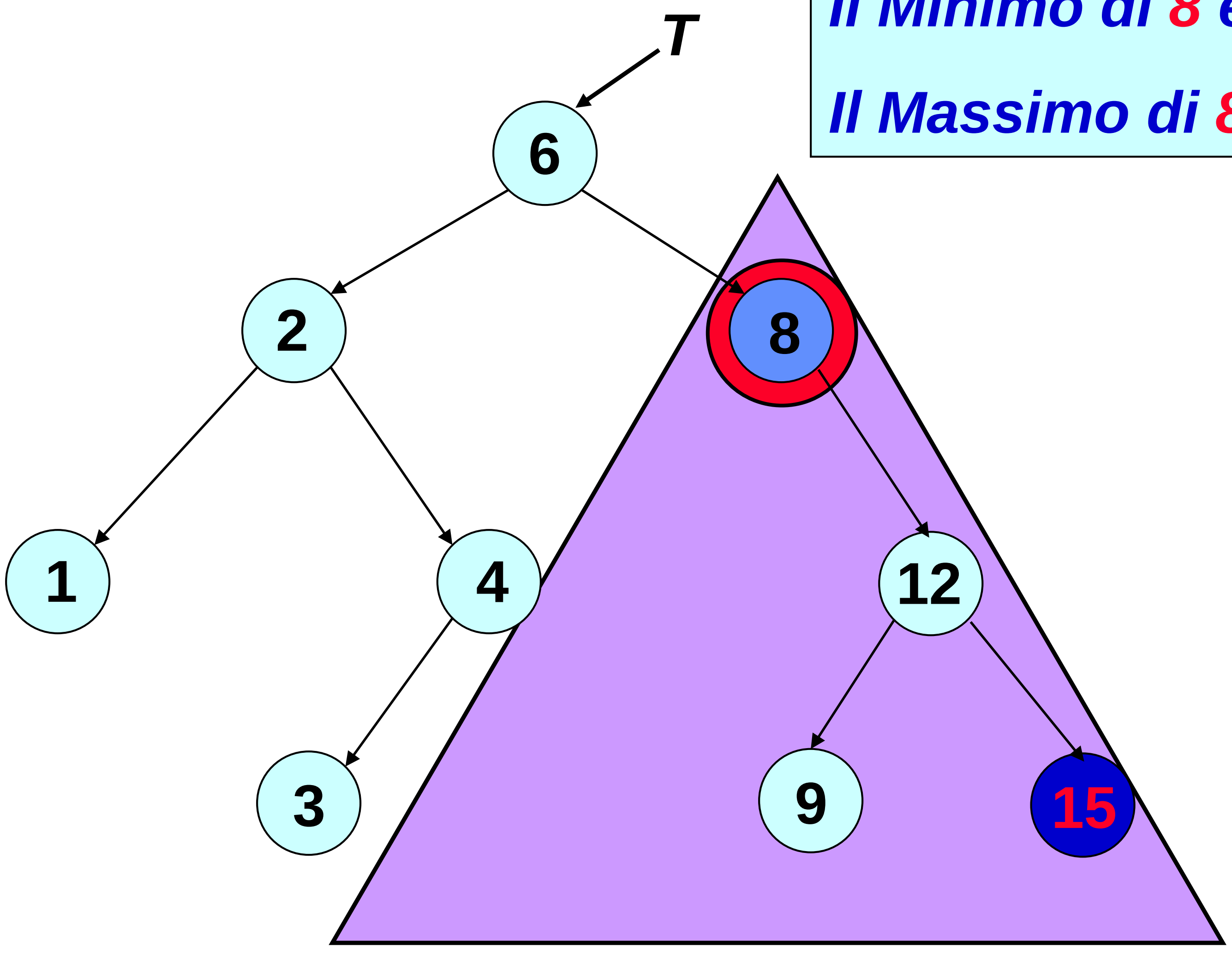
# ARB: ricerca del minimo e massimo

*Il Minimo di 2 è 1*

*Il Massimo di 2 è 4*



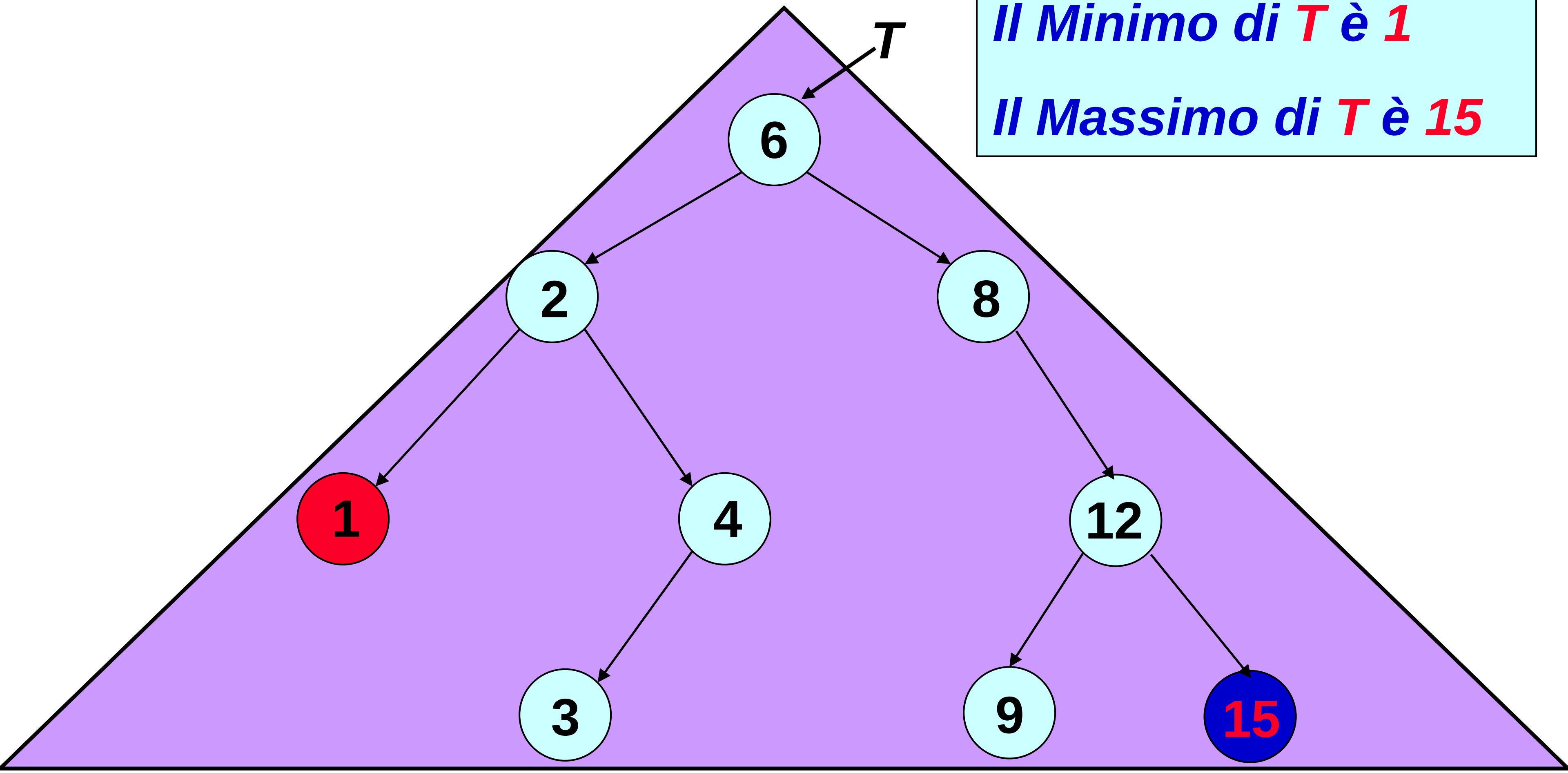
# *ARB: ricerca del minimo e massimo*




***Il Minimo di 8 è 8***

***Il Massimo di 8 è 15***

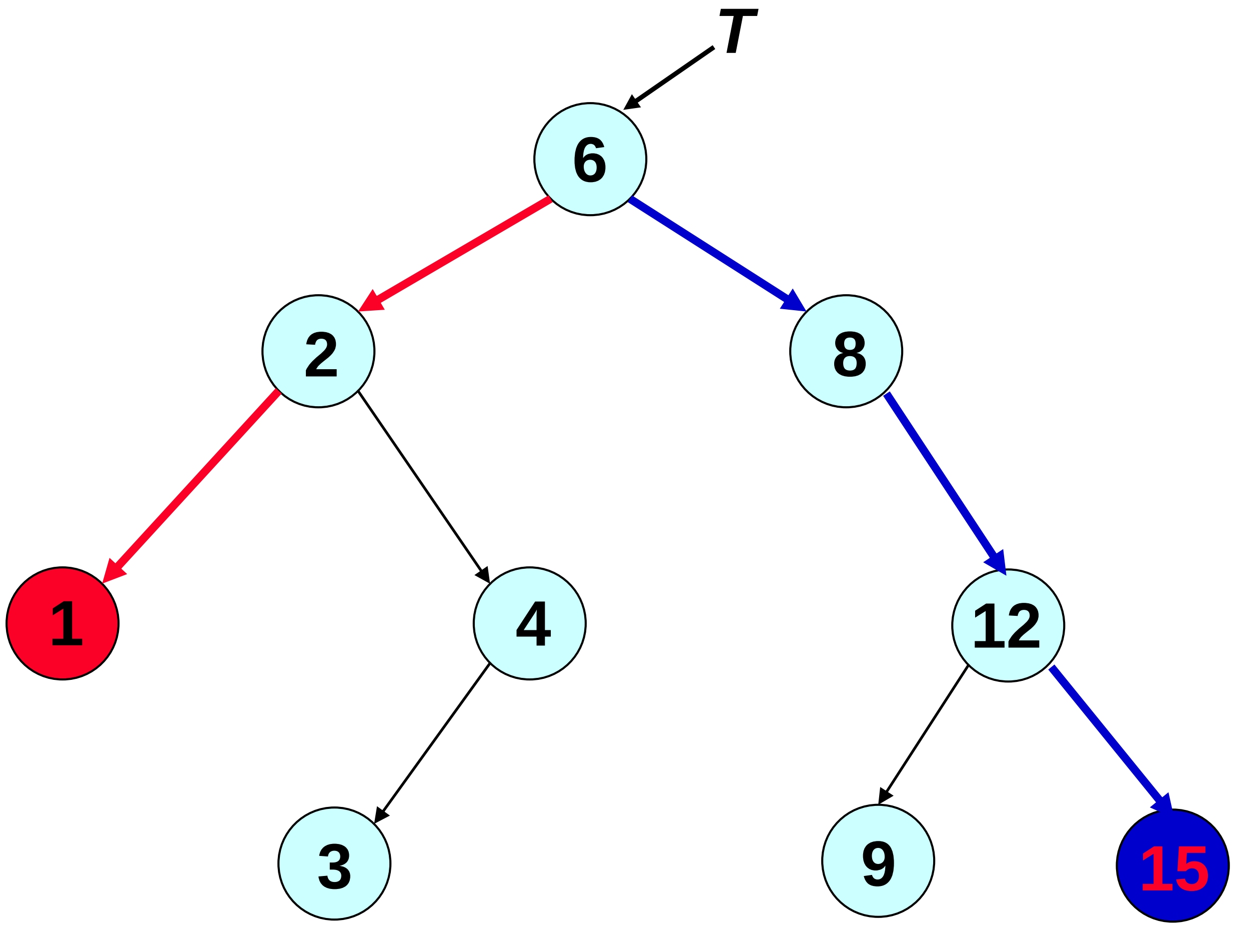
# ARB: ricerca del minimo e massimo




***Il Minimo di T è 1***

***Il Massimo di T è 15***

# ARB: ricerca del minimo e massimo



# *ARB: ricerca del minimo e massimo*

```
ARB ABR-Minimo(x:ARB)
   WHILE x->sx ≠ NIL DO
        x = x->sx
   return x
```

```
ARB ABR-Massimo(x: ARB)
   WHILE x->dx ≠ NIL DO
        x = x->dx
   return x
```

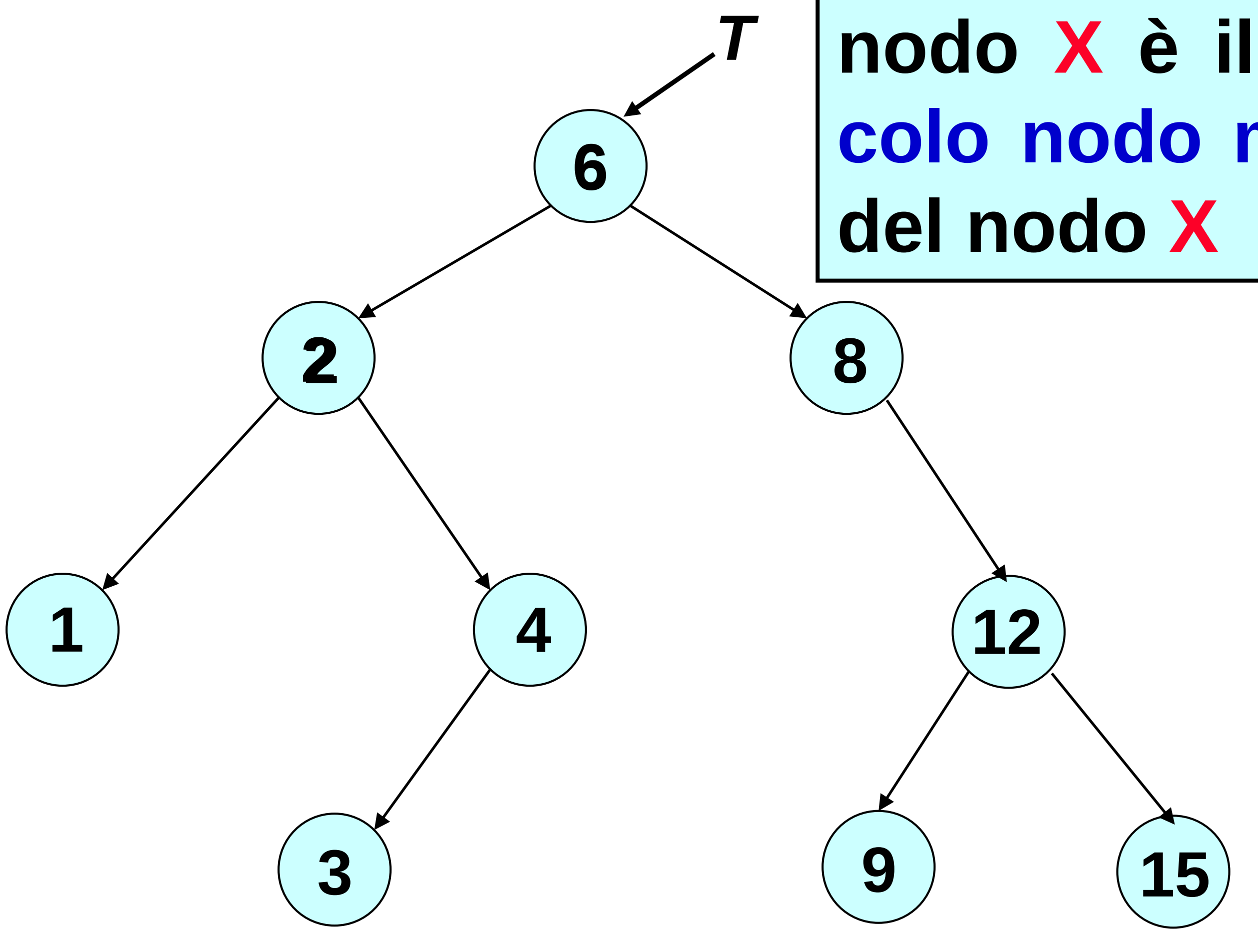
# *ARB: ricerca del minimo e massimo*

```
ARB ABR-Minimo(x:ARB)
    WHILE x->sx ≠ NIL DO
          x = x->sx
    return x
```

```
ARB ABR-Massimo(x: ARB)
    WHILE x->dx ≠ NIL DO
          x = x->dx
    return x
```

```
ARB ARB_Minimo(x:ARB)
   IF x ≠ NIL AND x->sx ≠ NIL THEN
        return ARB_Minimo(x->sx)
   return x
```

# *ARB: ricerca del successore*

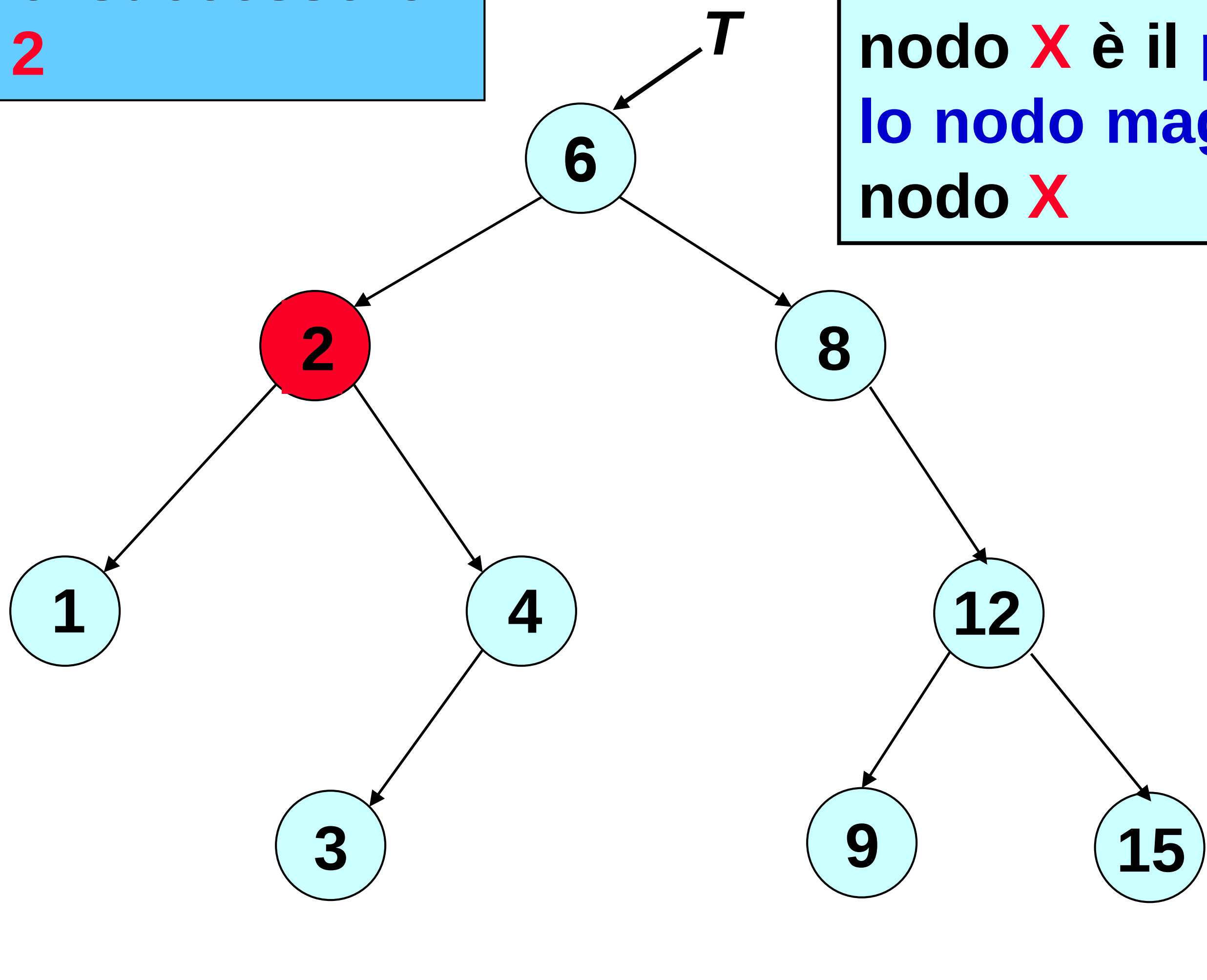



**Il successore di un nodo X è il più piccolo nodo maggiore del nodo X**

# *ARB: ricerca del successore*

***Ricerca*** **del successore del nodo 2**

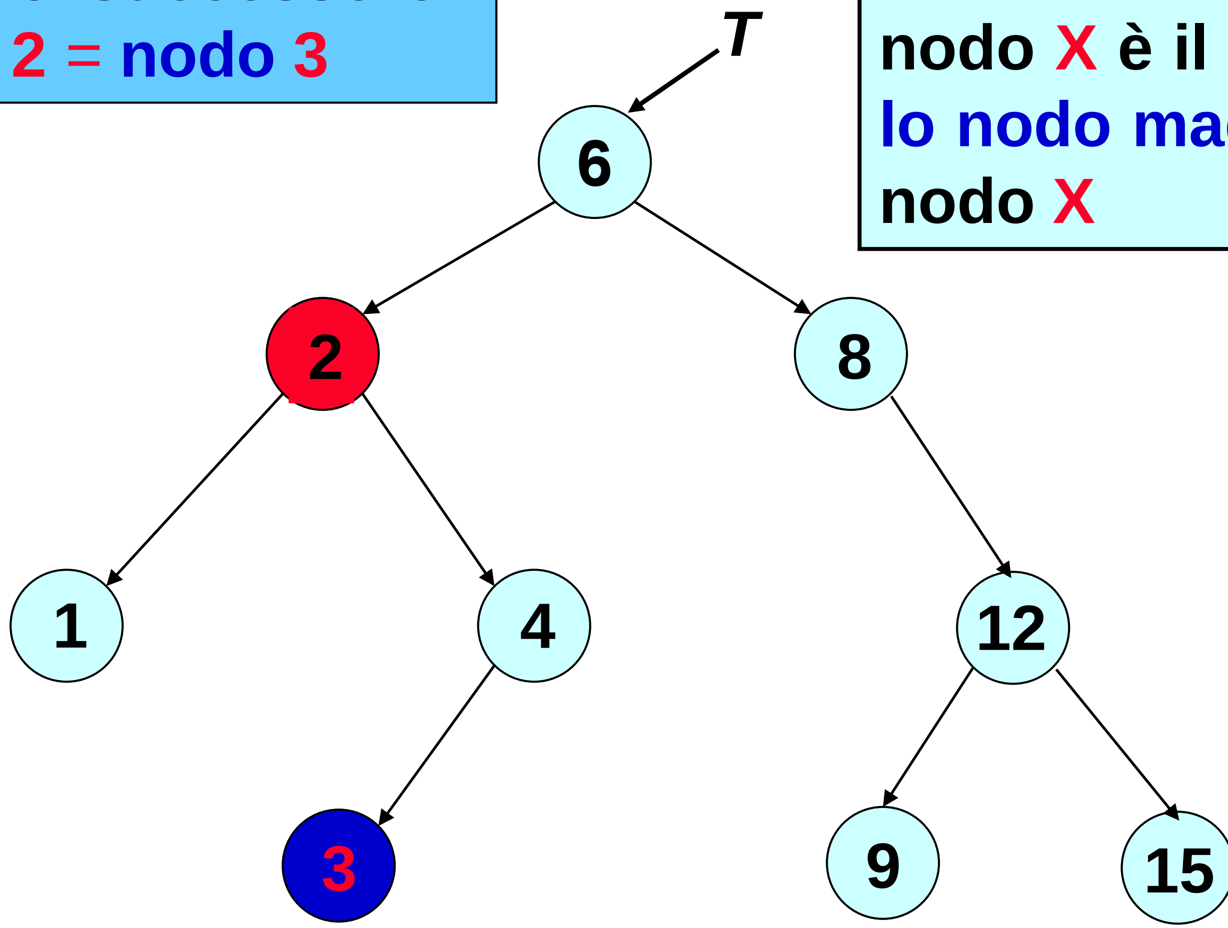
**Il successore di un nodo X è il più piccolo nodo maggiore del nodo X**



# *ARB: ricerca del successore*

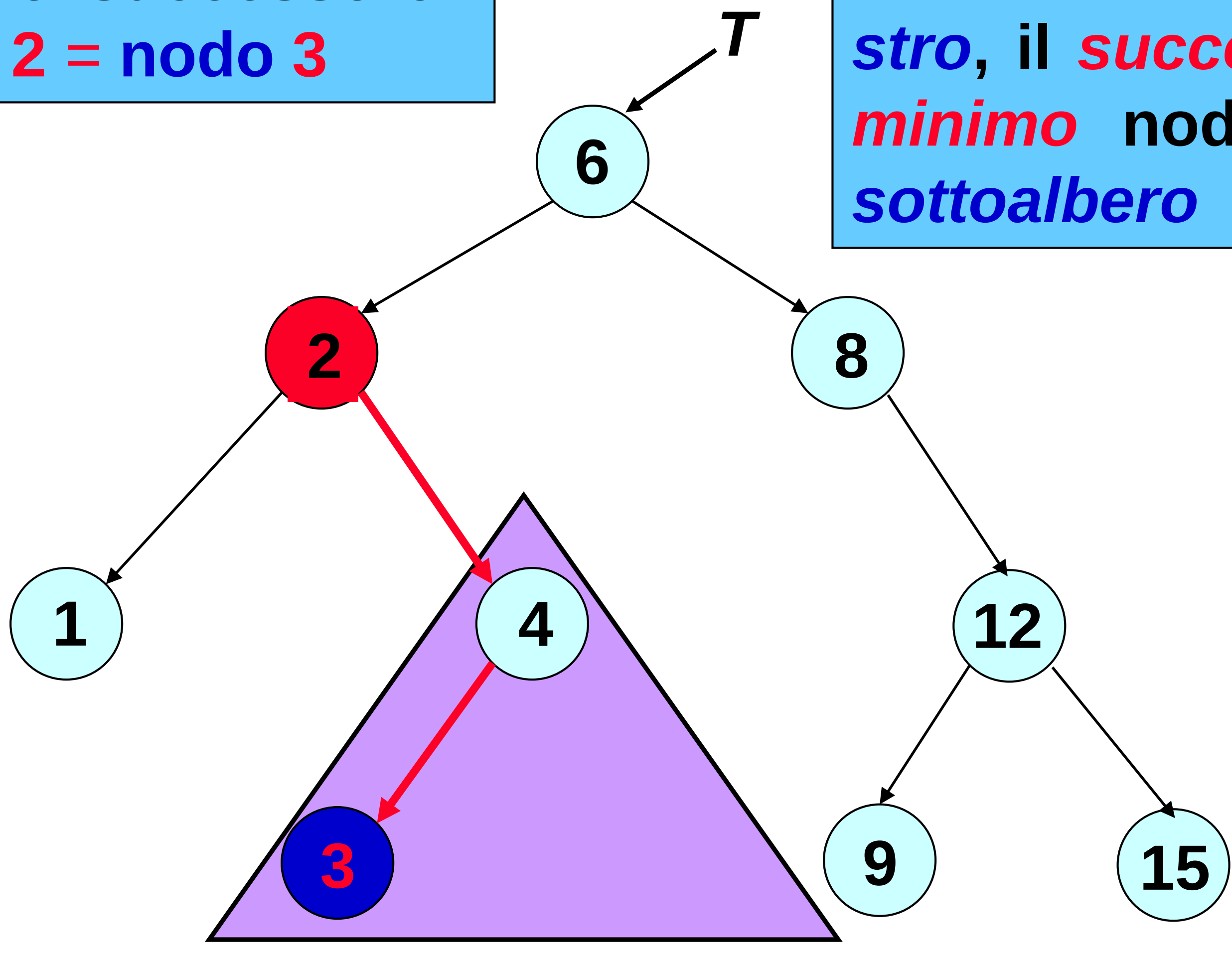
***Ricerca*** **del successore del nodo 2 = nodo 3**

**Il successore di un nodo X è il più piccolo nodo maggiore del nodo X**



# *ARB: ricerca del successore*

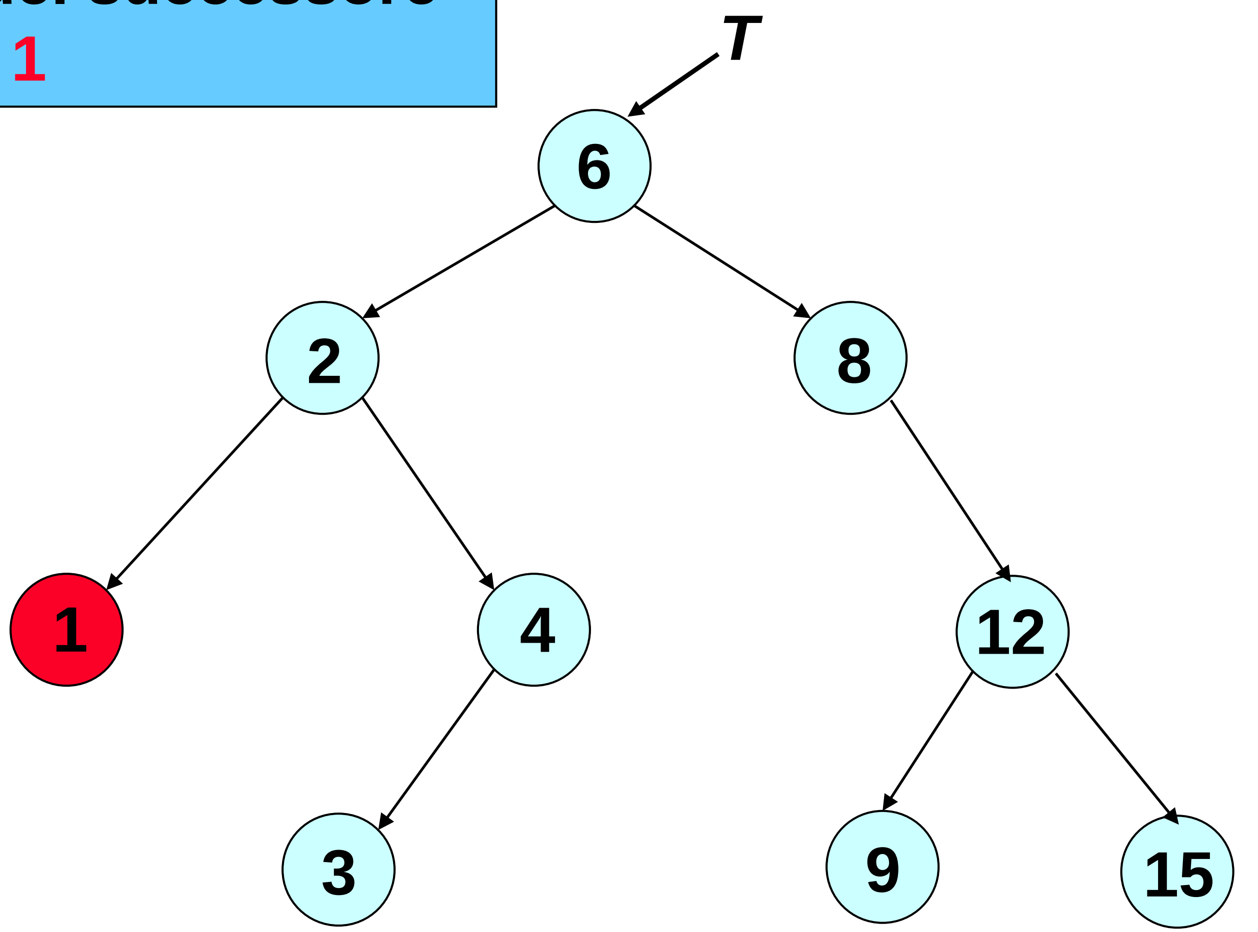
*Ricerca* **del successore del nodo 2 = nodo 3**




**Se *x* ha un *figlio destro*, il *successore* è il *minimo* nodo di quel *sottoalbero***

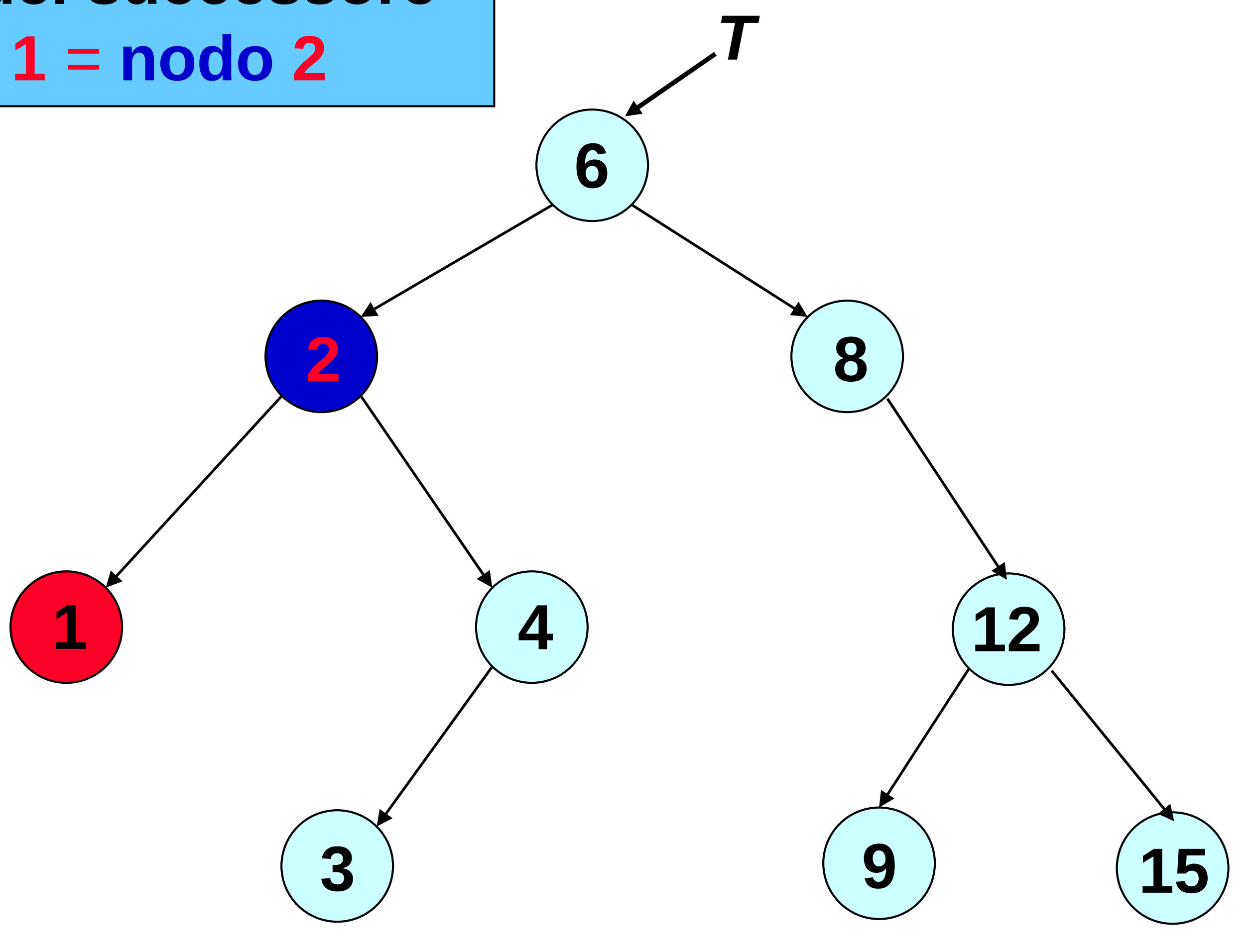
# ARB: ricerca del successore

*Ricerca* del successore del nodo 1



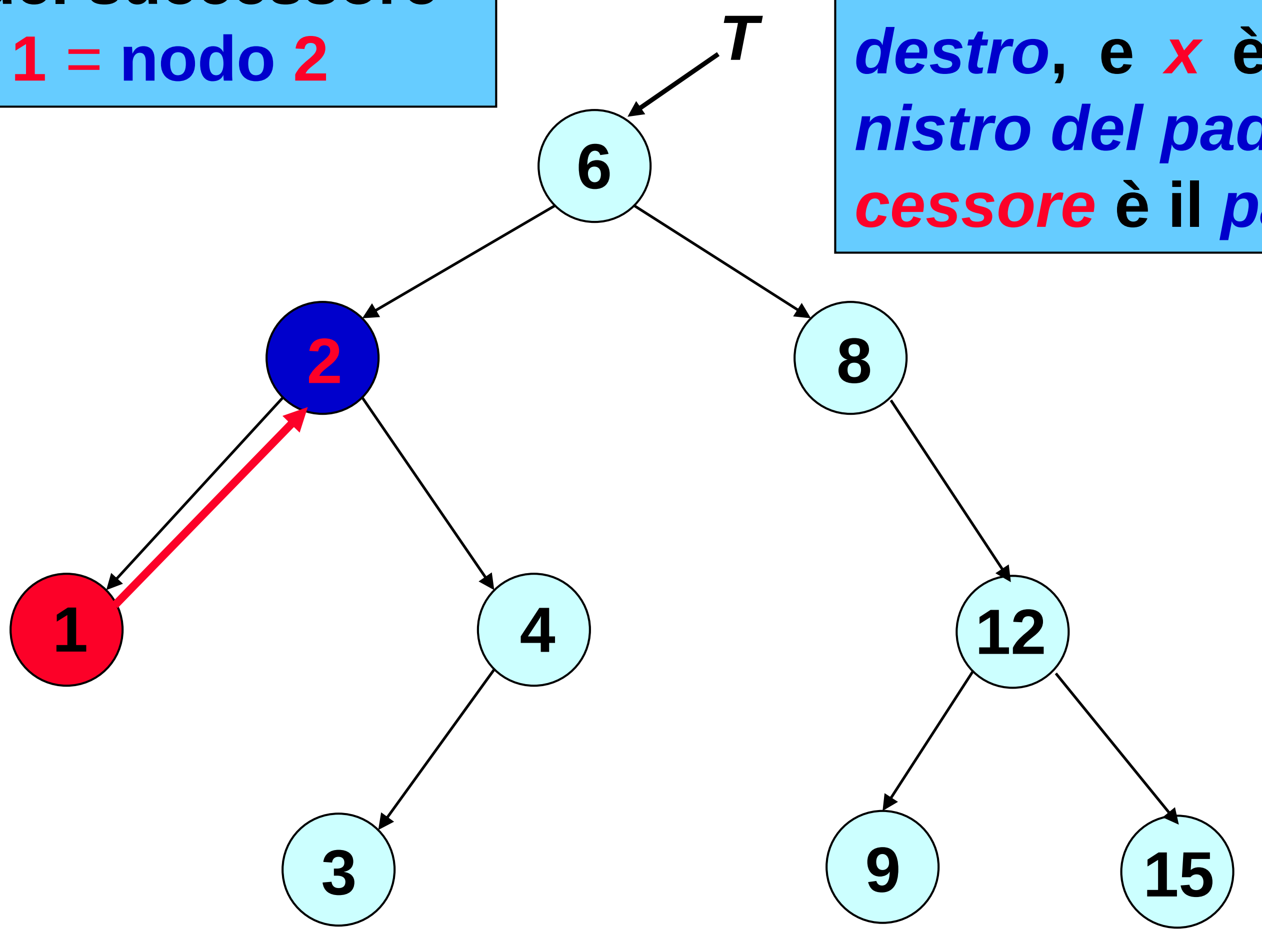
# ARB: ricerca del successore

*Ricerca* del successore
del nodo 1 = nodo 2



# *ARB: ricerca del successore*

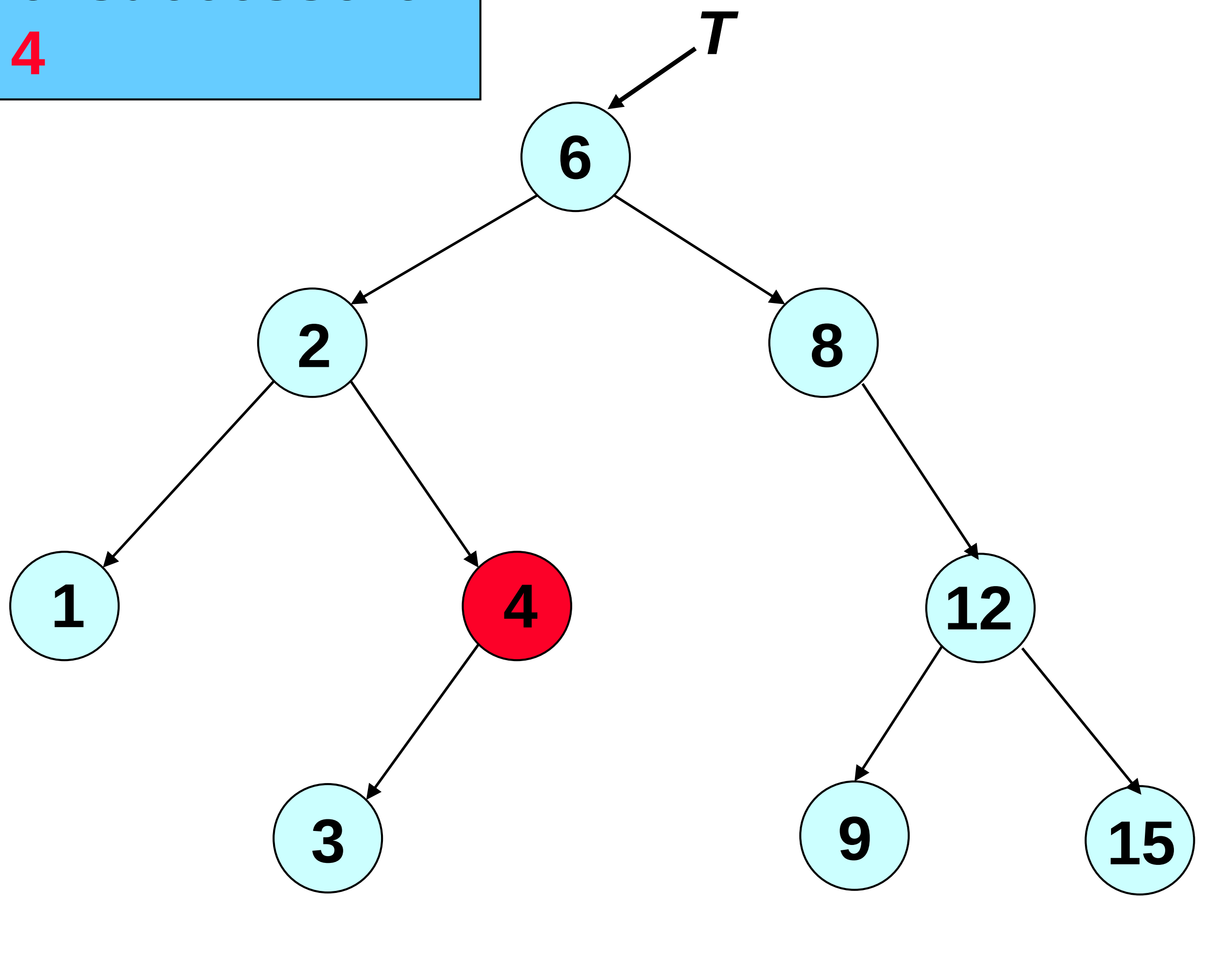
***Ricerca*** **del successore del nodo 1 = nodo 2**




**Se *x* NON ha un *figlio destro*, e *x* è *figlio sinistro del padre*, il *successore* è il *padre*.**

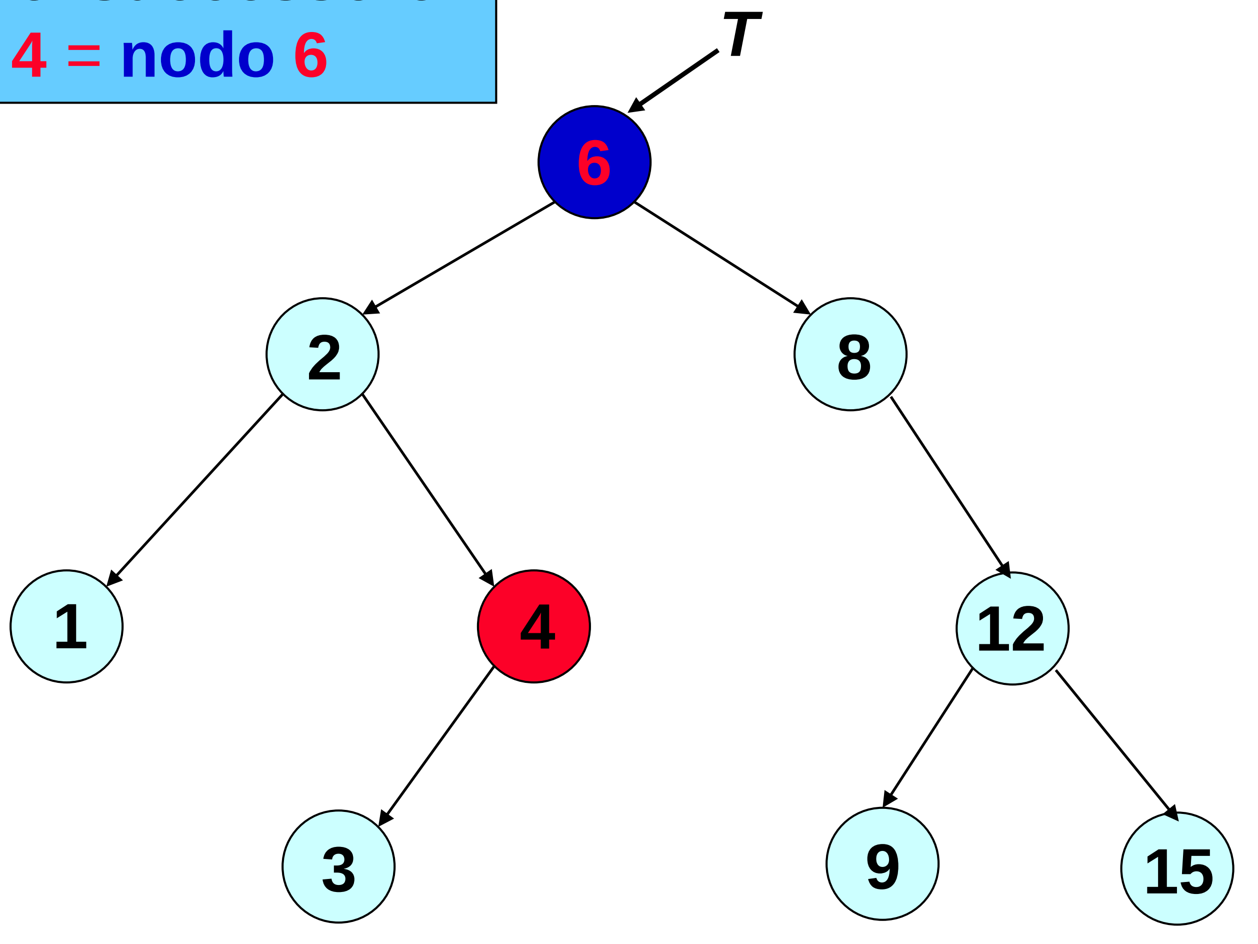
# *ARB: ricerca del successore*

***Ricerca* del successore del nodo 4**



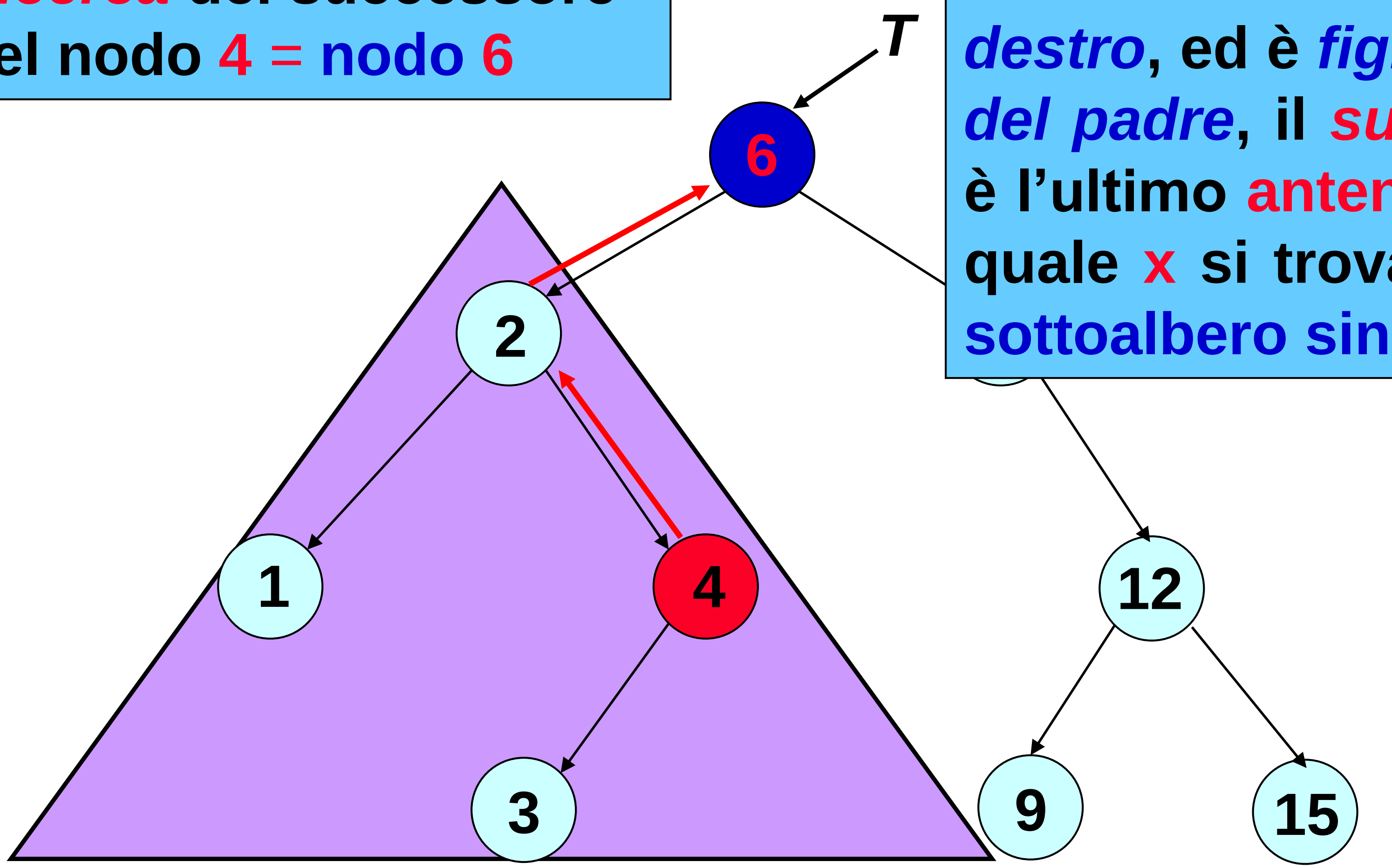
# ARB: ricerca del successore

*Ricerca* del successore
del nodo 4 = nodo 6



# *ARB: ricerca del successore*

***Ricerca*** **del successore del nodo 4 = nodo 6**




**Se *x* NON ha un *figlio destro*, ed è *figlio destro del padre*, il *successore* è l'ultimo antenato per il quale x si trova nel suo sottoalbero sinistro.**

# ARB: ricerca del successore

```
ABR-Successore(T,k)
   Z = T
   P = NIL
   WHILE (Z ≠ NIL && Z->key ≠ k)
         P = T
         IF (Z->key < k) THEN
              Z = Z->dx
         ELSE
              Z = Z->sx
   IF (Z ≠ NIL && Z->dx ≠ NIL) THEN
             return ABR-Minimo(Z->dx)
   ELSE
         WHILE (P ≠ NIL && (Z ≠ P->sx || P->key < k)) DO
              Z = P
              P = Z->padre
         return P
```

**Se *Z* NON ha un *figlio destro*, ed è *figlio destro del padre*, il *successore* è l'ultimo antenato per il quale Z si trova nel suo sottoalbero sinistro.**

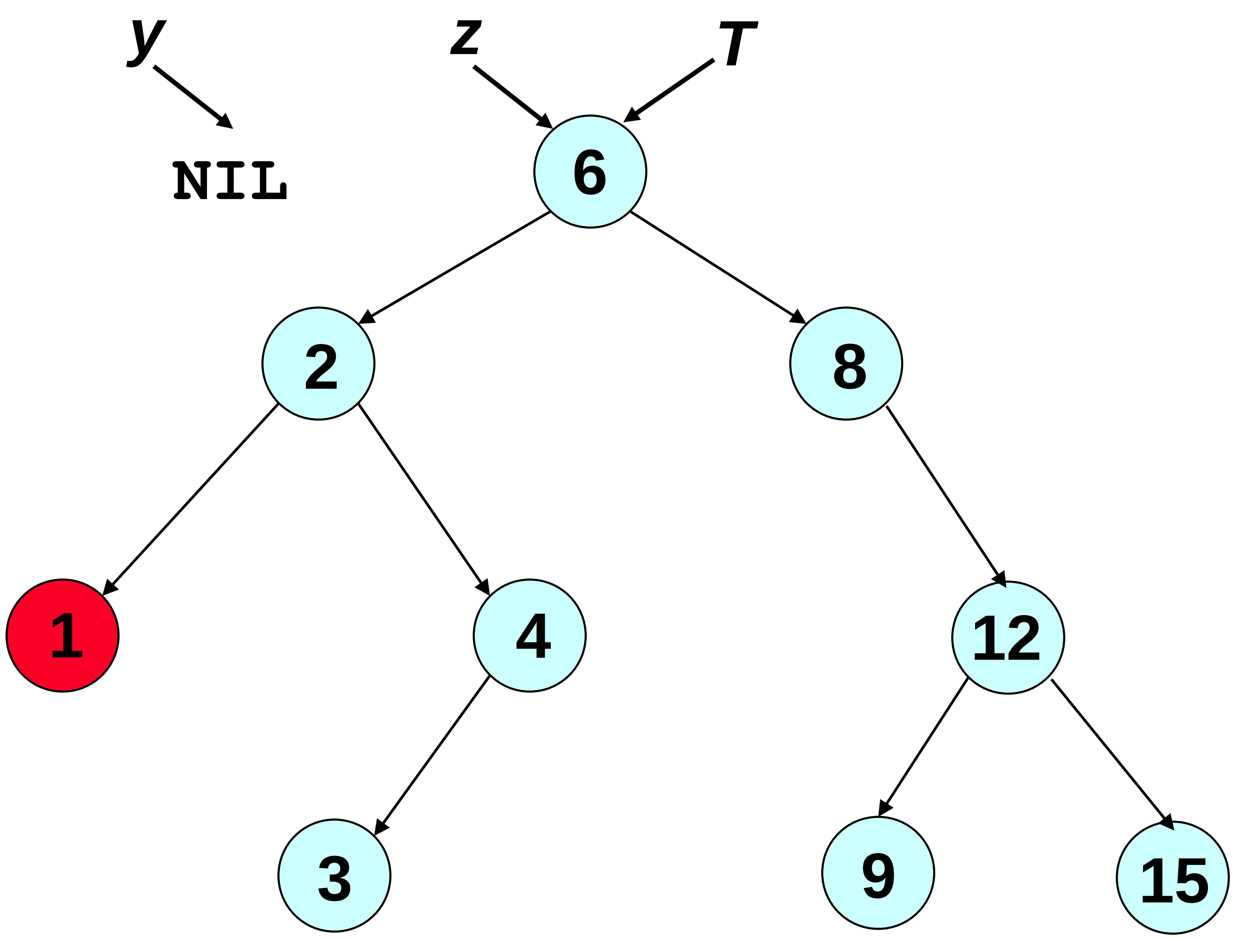
# ARB: ricerca del successore

```
ABR-Successore(T,k)
   Z = T
   P = NIL
   WHILE (Z ≠ NIL && Z->key ≠ k)
         P = T
         IF (Z->key < k) THEN
             Z = Z->dx
```

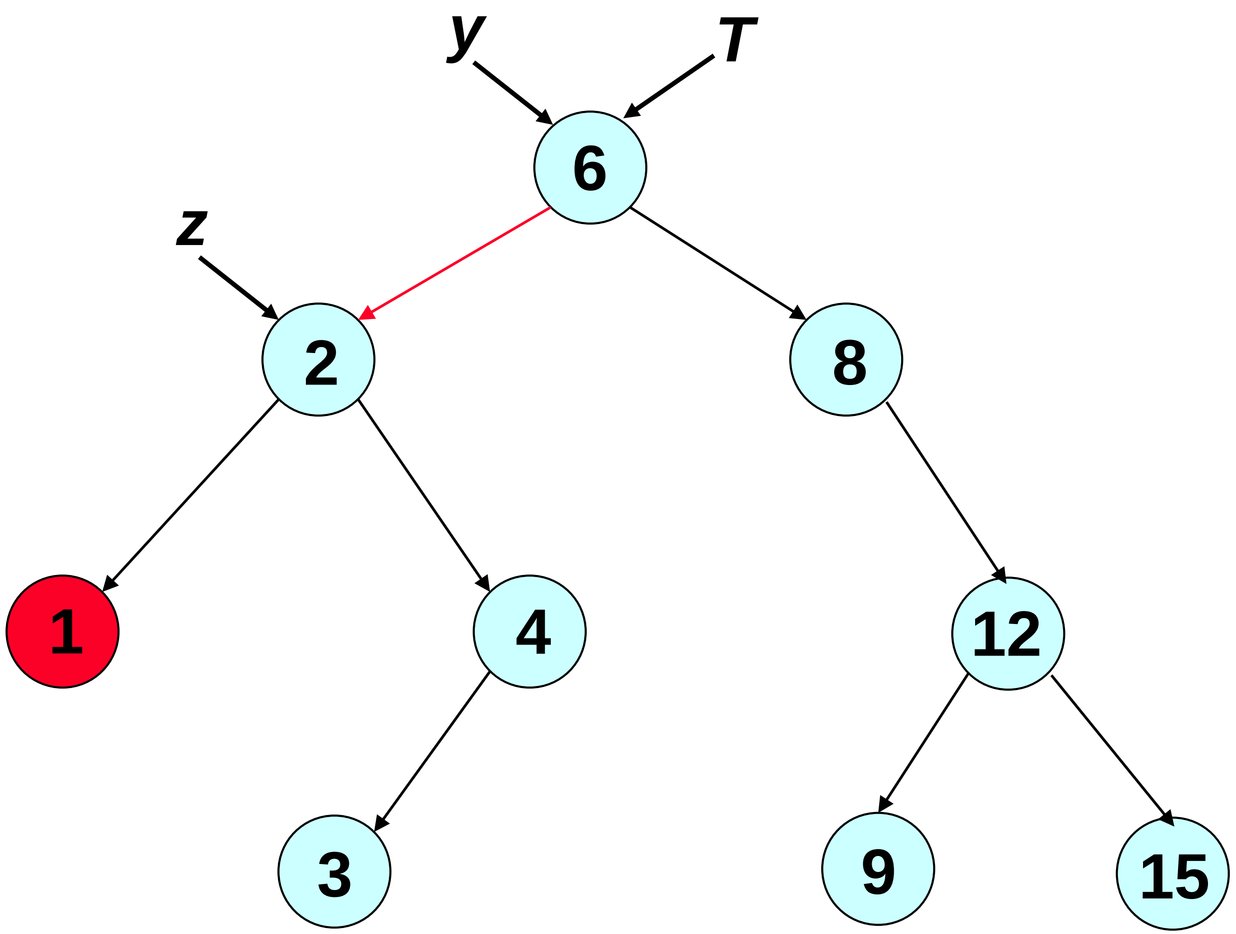
**Questto algoritmo assume che ogni nodo abbia il puntatore al padre**

```
   IF (Z ≠ NIL && Z->dx ≠ NIL) THEN
            return ABR-Minimo(Z->dx)
   ELSE
        WHILE (P ≠ NIL && (Z ≠ P->sx || P->key < k)) DO
              Z = P
              P = Z->padre
        return P
```

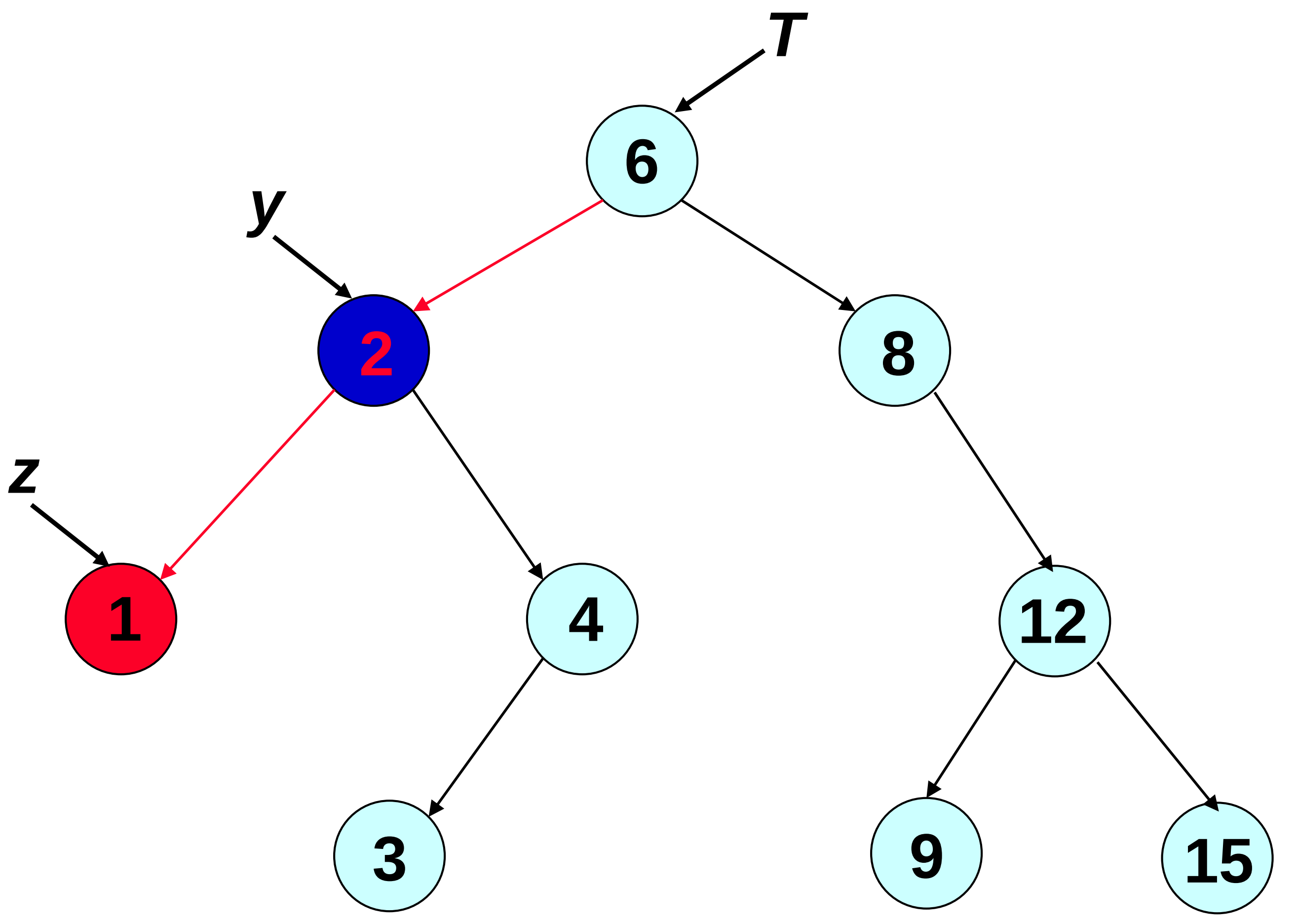
# ARB: ricerca del successore II



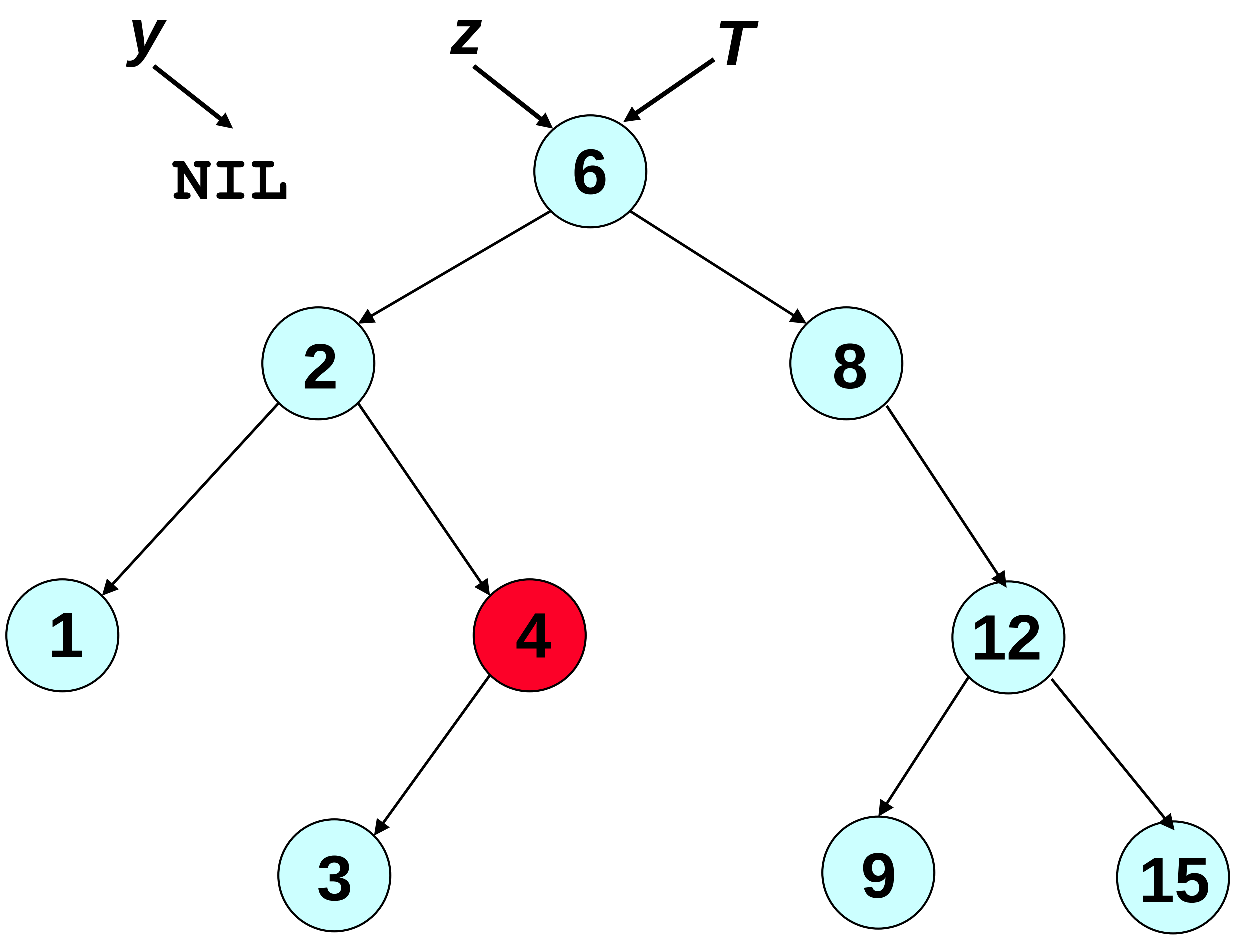
# ARB: ricerca del successore II



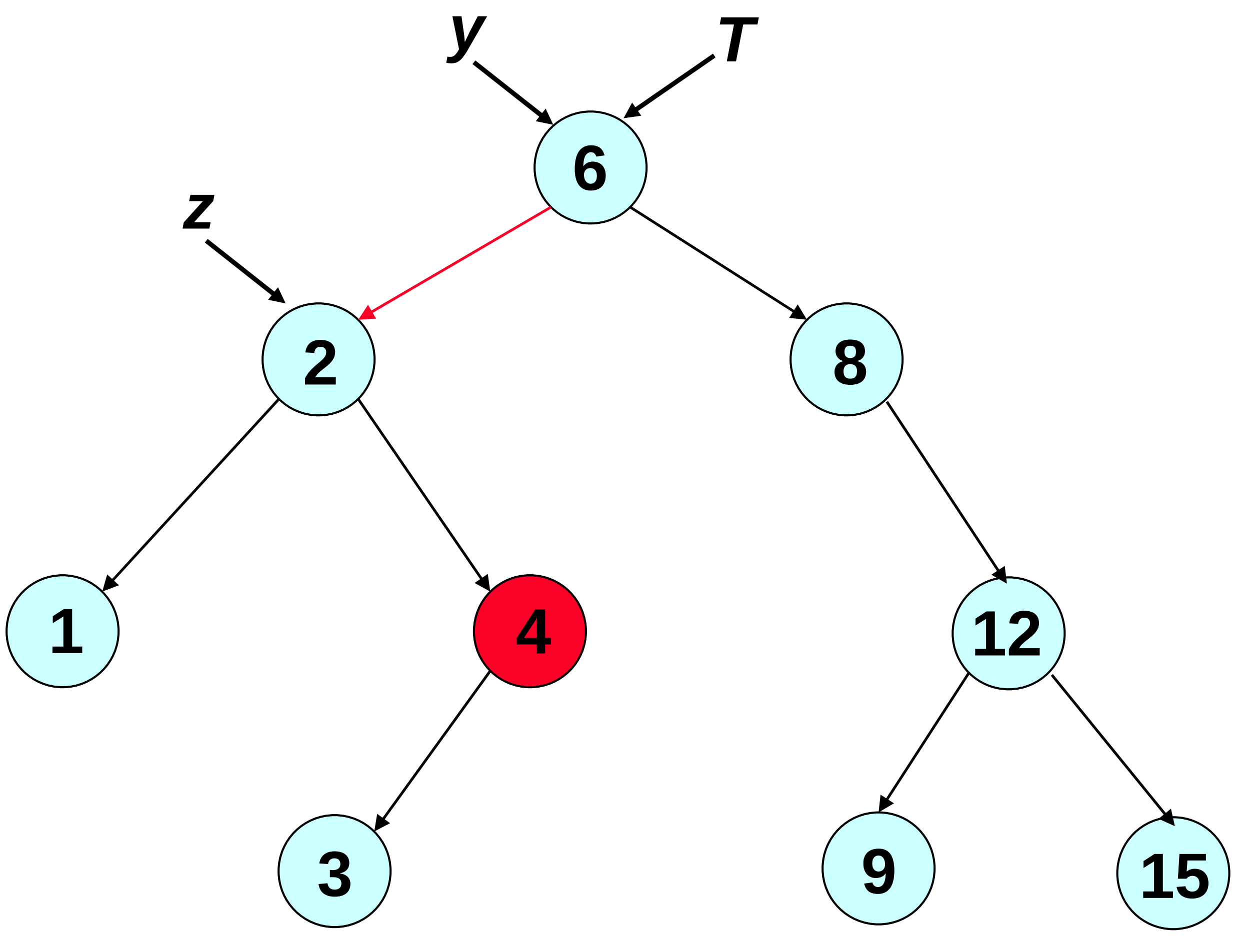
# ARB: ricerca del successore II



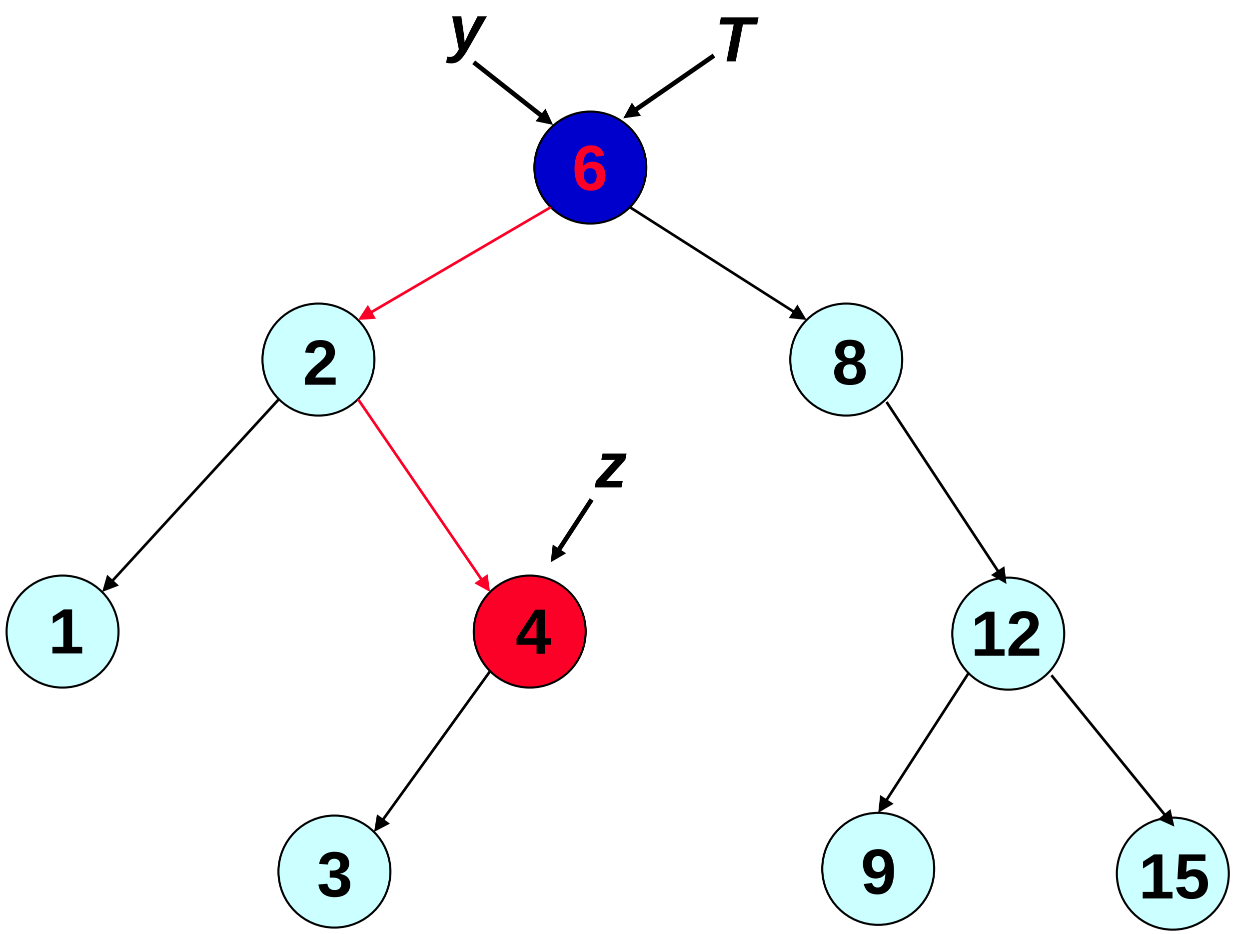
# ARB: ricerca del successore II



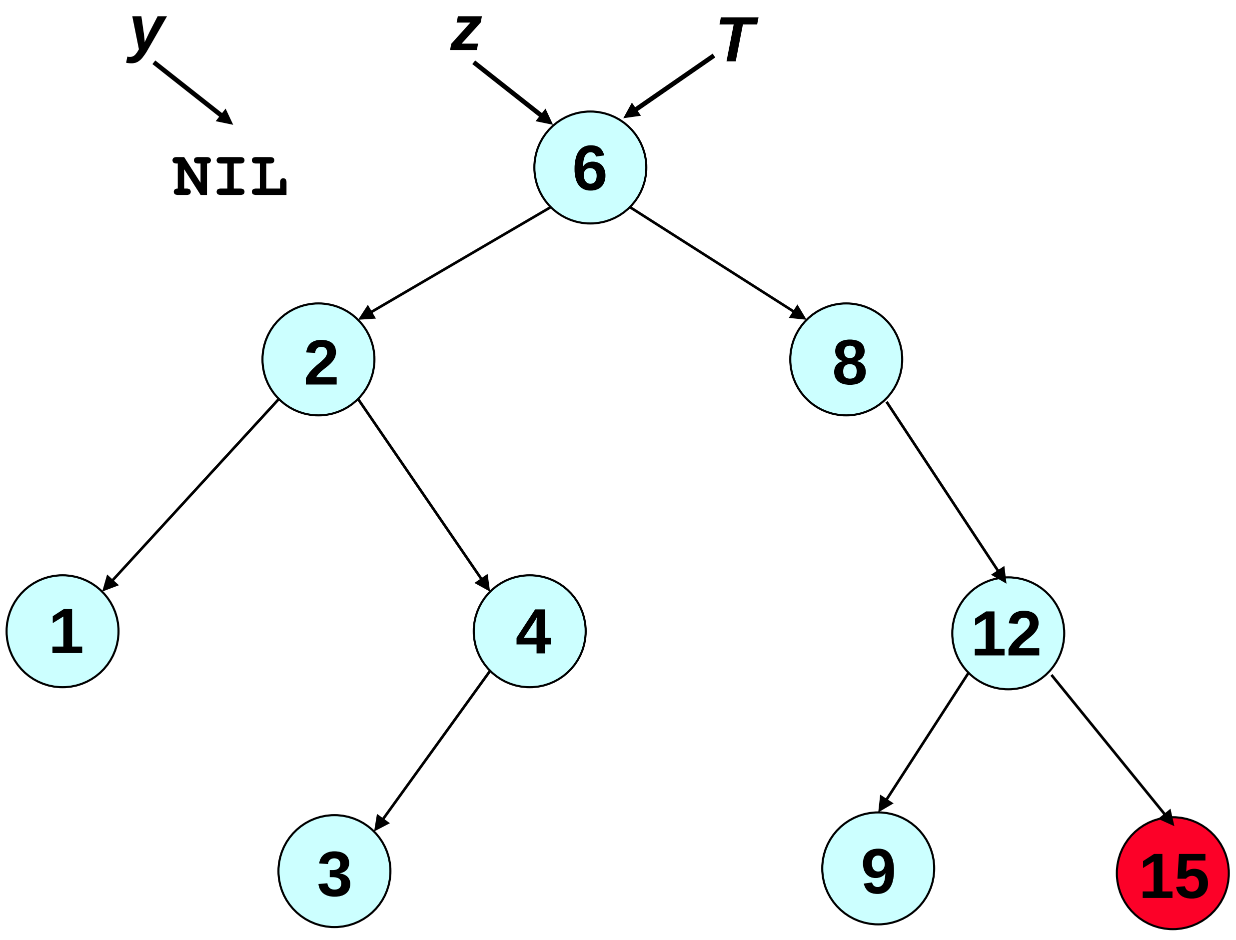
# ARB: ricerca del successore II



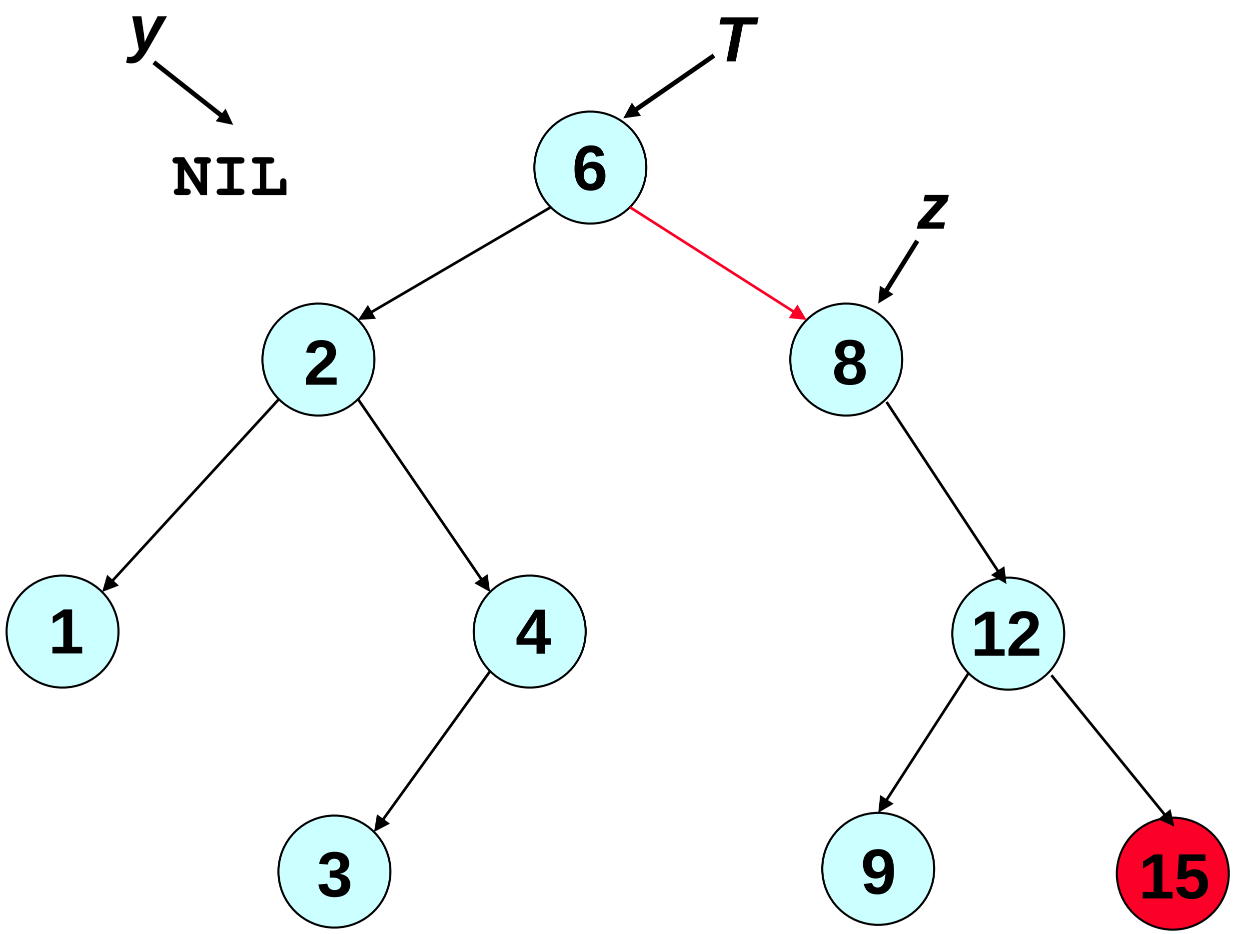
# ARB: ricerca del successore II



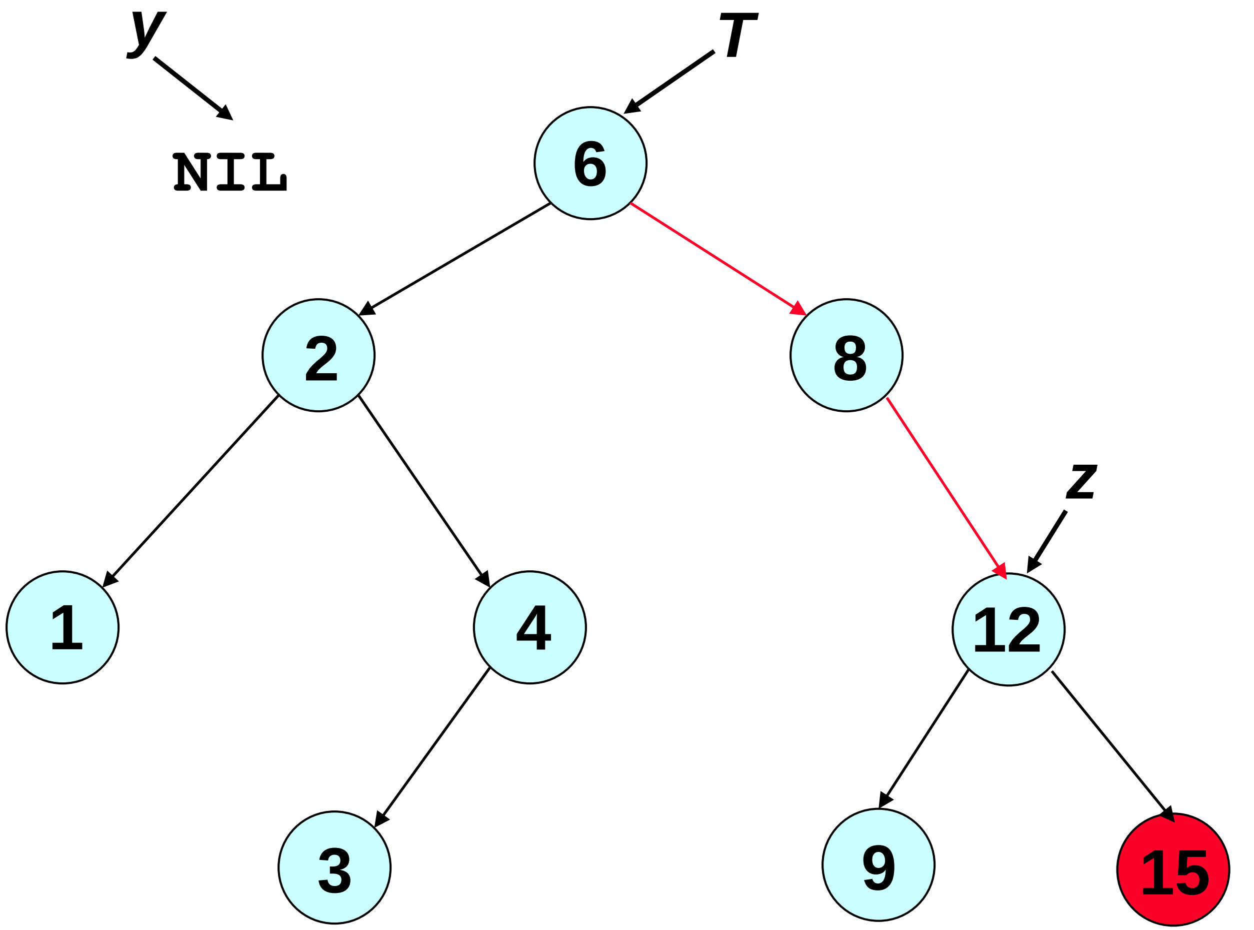
# ARB: ricerca del successore II



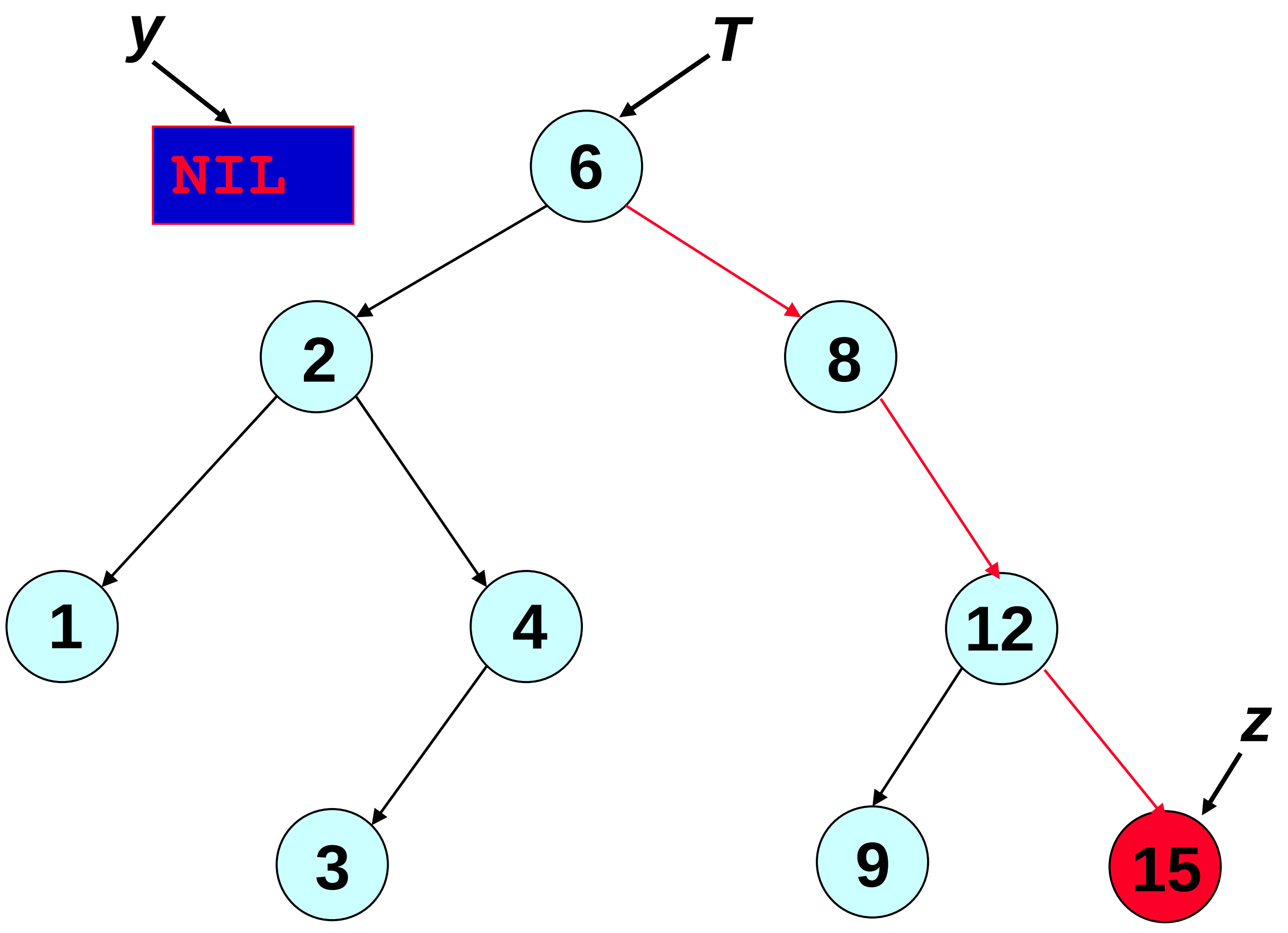
# ARB: ricerca del successore II



# *ARB: ricerca del successore II*



# ARB: ricerca del successore II



## *ARB: ricerca del successore II*

- **Inizializziamo il *successore* a *NIL***

- **Partendo dalla radice dell’albero:**
  - **ogni volta che si segue il *ramo sinistro* per raggiungere il nodo, si aggiorna il successore al nodo padre;**
  - **ogni volta che si segue un *ramo destro* per rag-giungere il nodo, NON si aggiorna il successore al nodo padre;**

# ARB: ricerca del successore ricorsiva

```
ABR-Successore_ric(T,k)
  IF (T ≠ NIL) THEN
    IF (T->key = k)
         return ABR-Minimo(T->dx)
    ELSE IF (T->key < k) THEN
         return ABR-Successore_ric(T->dx,k)
    ELSE /* key[T] > key */
         succ = ABR-Successore_ric(T->sx,k)
         IF (succ ≠ NIL) THEN
             return succ
  return T
```

# ARB: ricerca del successore

```
ARB ABR-Successore'(T, k)
   z = T
   y = NIL
   WHILE (z ≠ NIL && z->key ≠ k)
      IF (z->key < k)
            z = z->dx
      ELSE IF (z->key > k)
            y = z
            z = z->sx
   IF (z ≠ NIL && z->dx ≠ NIL) THEN
        y = ABR-Minimo(z->dx)
   return y
```

**y punta sempre al miglior candidato a successore**

# ARB: ricerca del successore ricorsiva

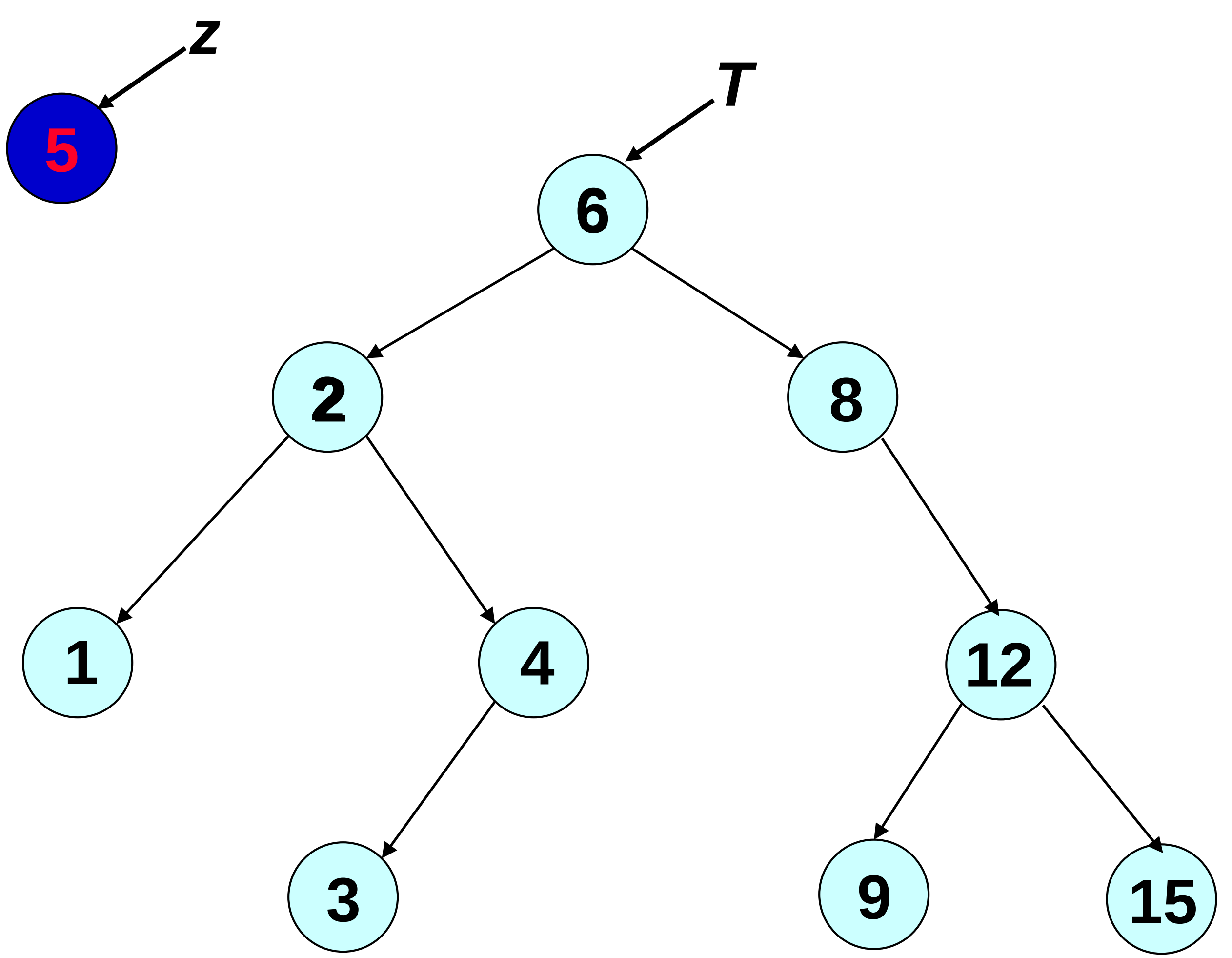
```
ABR-Successore_ric'(T,k,y)
  IF (T ≠ NIL) THEN
    IF (k > T->key) THEN
      return ABR-Successore_ric'(T->dx,k,y)
    ELSE IF (k < T->key) THEN
      return ABR-Successore_ric'(T->sx,k,T)
    ELSE /* k = T->key */
      IF (T->dx ≠ NIL) THEN
        return ABR-Minimo(T->dx)
  return y
```

```
ABR-Successore'(T,k)
   return ABR-Successore_ric'(T,k,NIL)
```

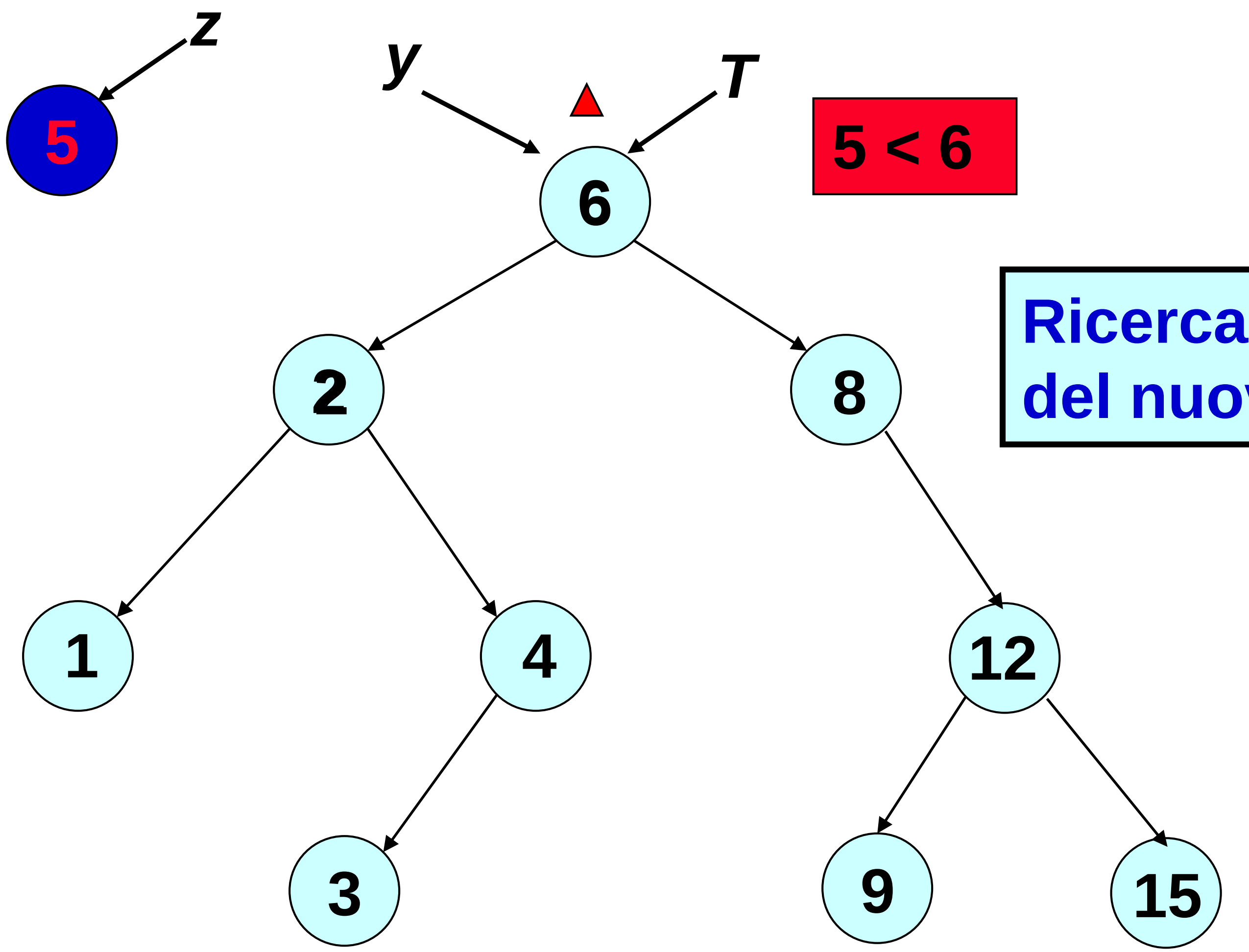
## *ARB: costo delle operazioni*

***Teorema**. Le operazioni di Ricerca, Minimo, Massimo, Successore e Predecessore su di un <u>Albero Binario di Ricerca</u> possono essere eseguite in tempo $O(h)$, dove $h$ è l'altezza dell'albero.*

# ARB: Inserimento di un nodo (caso I)



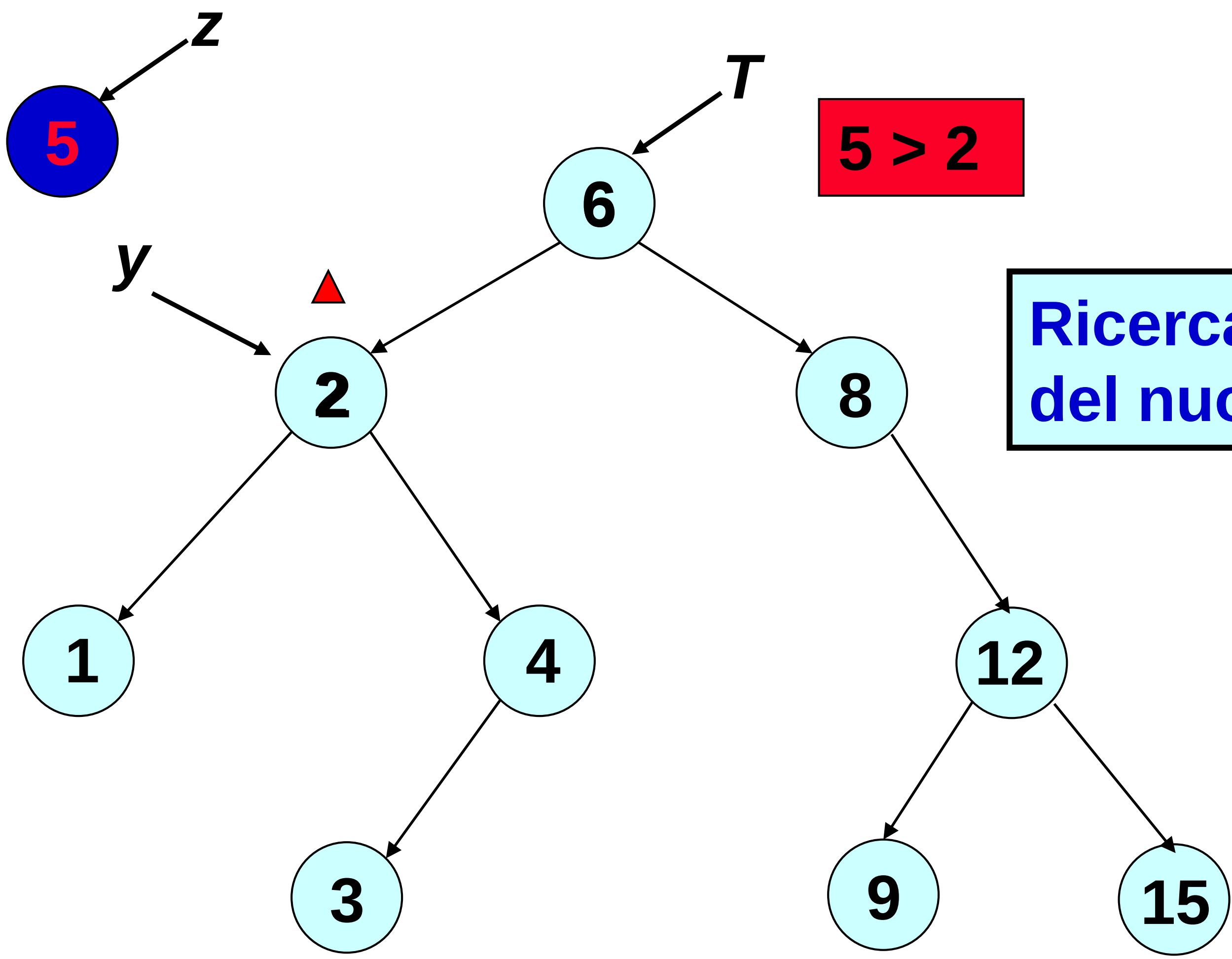
# *ARB: Inserimento di un nodo (caso I)*




**5 < 6**

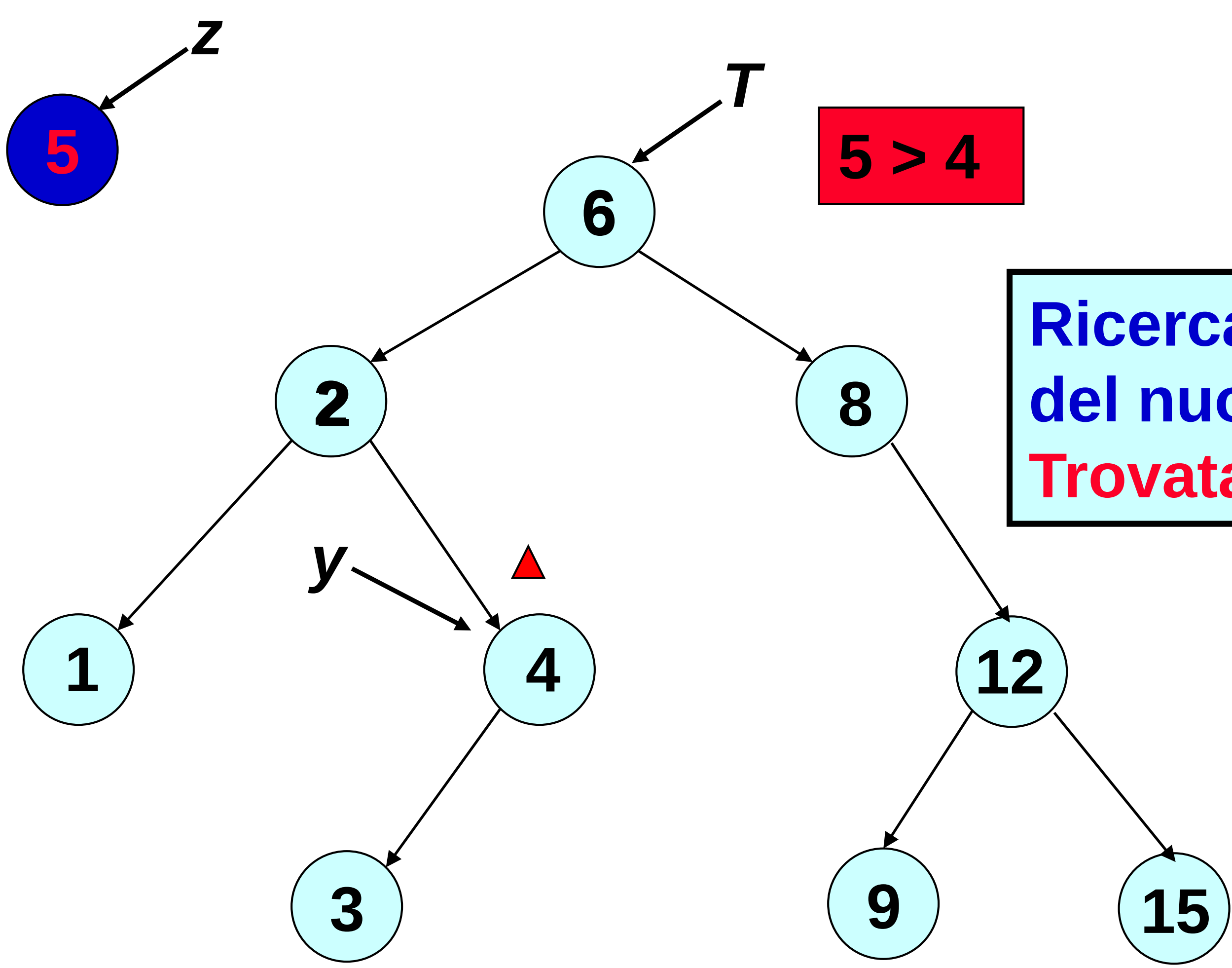
**Ricerca posizione del nuovo nodo**

# ARB: Inserimento di un nodo (caso I)



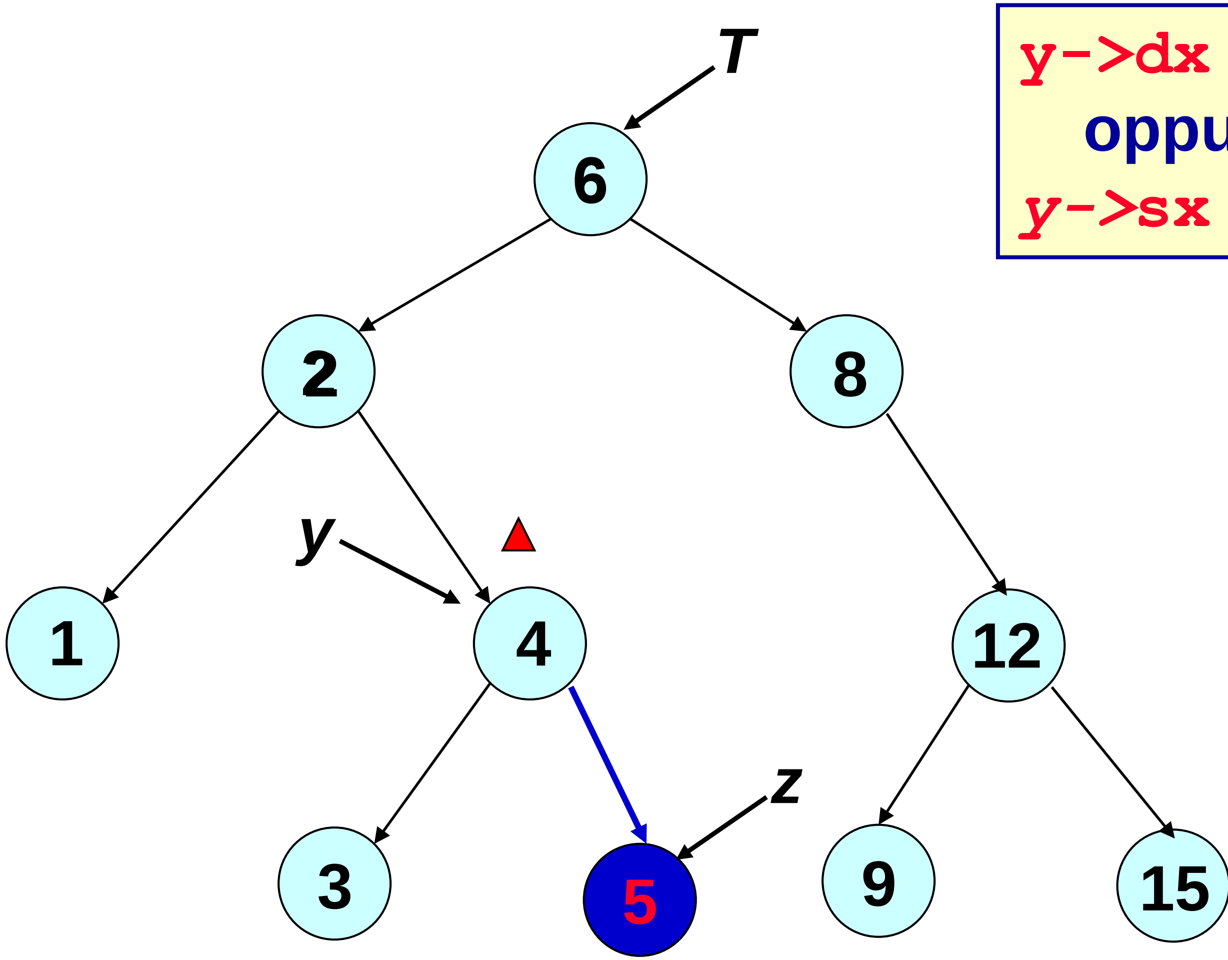

Ricerca posizione
del nuovo nodo

# *ARB: Inserimento di un nodo (caso I)*




**Ricerca posizione del nuovo nodo Trovata!**

# *ARB: Inserimento di un nodo (caso I)*

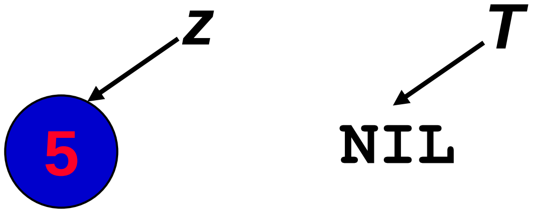



`y->dx = z`

**oppure**

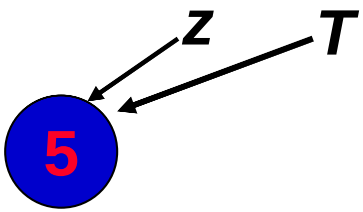
`y->sx = z`

# *ARB: Inserimento di un nodo (caso II)*




**Albero è vuoto**

# *ARB: Inserimento di un nodo (caso II)*




`Root[T] = z`

**Albero è vuoto**

**Il nuovo nodo da inserire diviene la radice**

# ARB: Inserimento di un nodo

```
ABR-inserisci(T,k)
   P = NIL
   x = T
   WHILE (x ≠ NIL) DO
        P = x
        IF (z->key < x->key) THEN
             x = x->sx
        ELSE
             x = x->dx
   z = alloca nodo ARB
   z->key = k
   IF P = NIL THEN
          T = z
   ELSE IF z->key < P->key THEN
             P->sx = z
        ELSE
             P->dx = z
```

# ARB: Inserimento di un nodo

```
ABR-inserisci(T,k)
   P = NIL
   x = T
   WHILE (x ≠ NIL) DO
        P = x
        IF (z->key < x->key) THEN
             x = x->sx
        ELSE
             x = x->dx
   z = alloca nodo ARB
   z->key = k
   IF P = NIL THEN
          T = z
   ELSE IF z->key < P->key THEN
             P->sx = z
        ELSE
             P->dx = z
```

Ricerca posizione del nuovo nodo

(caso II)

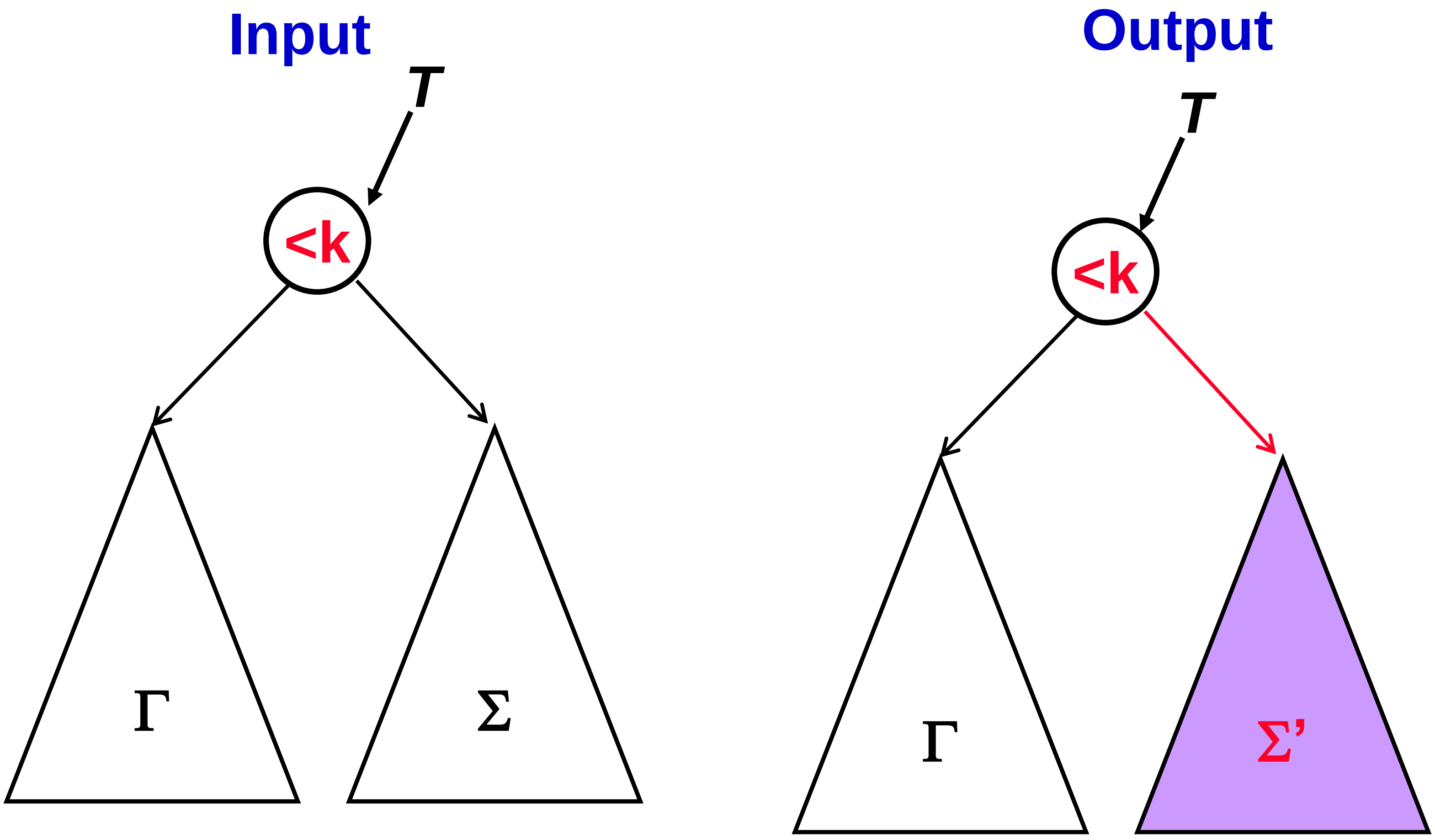
(caso I)

# *ARB: Inserimento di un nodo*

```
ABR-insert_ric(T,k)
 IF T ≠ NIL THEN
    IF k < T->key THEN
        T->sx = ABR-insert_ric(T->sx,k)
    ELSE
        T->dx = ABR-insert_ric(T->dx,k)
 ELSE
       T = alloca nodo ARB
       T->key = k
 return z
```

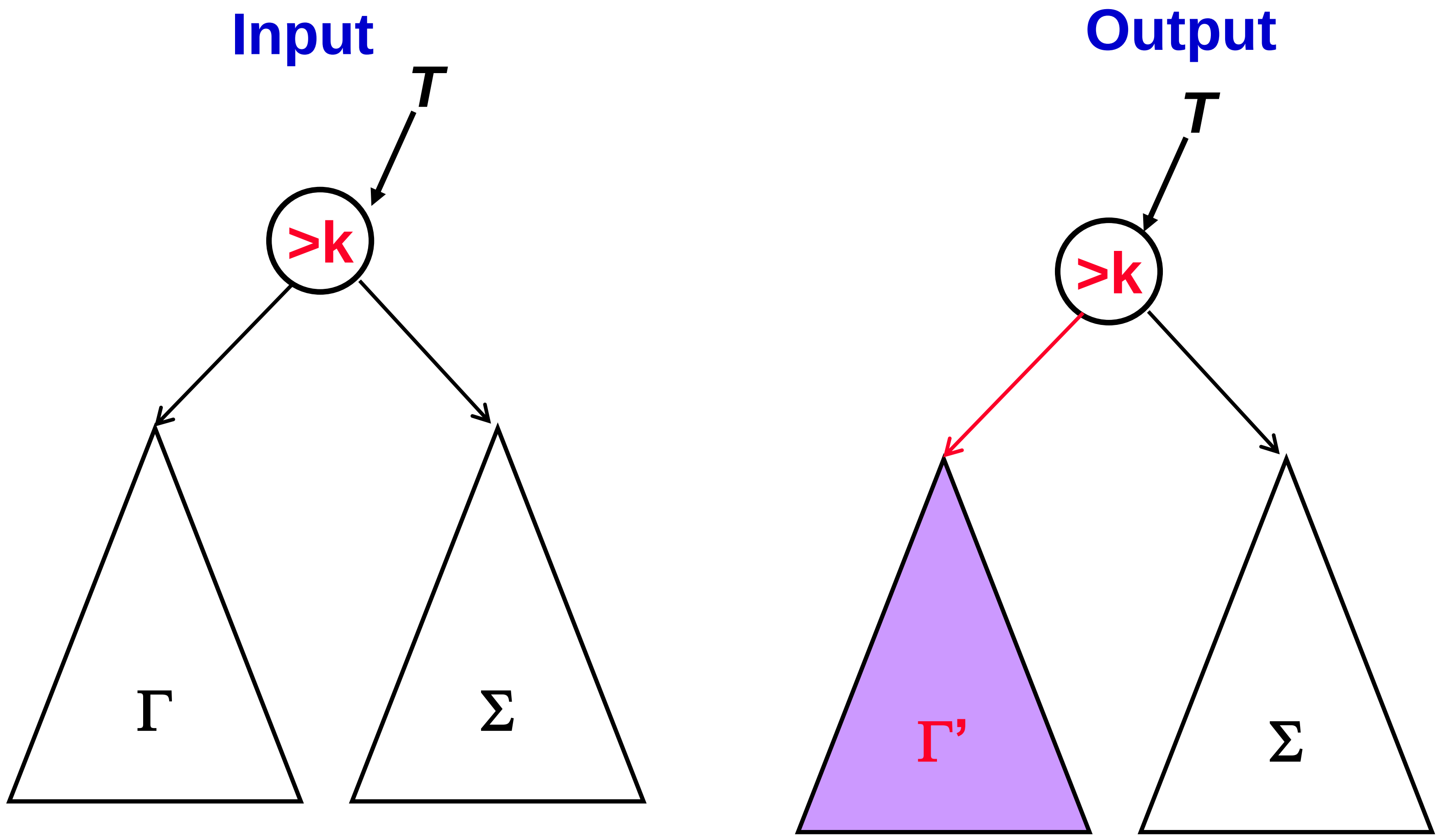
k è la chiave da inserire.

# *Cancellazione ricorsiva*




Σ’ = cancella(Σ,k)

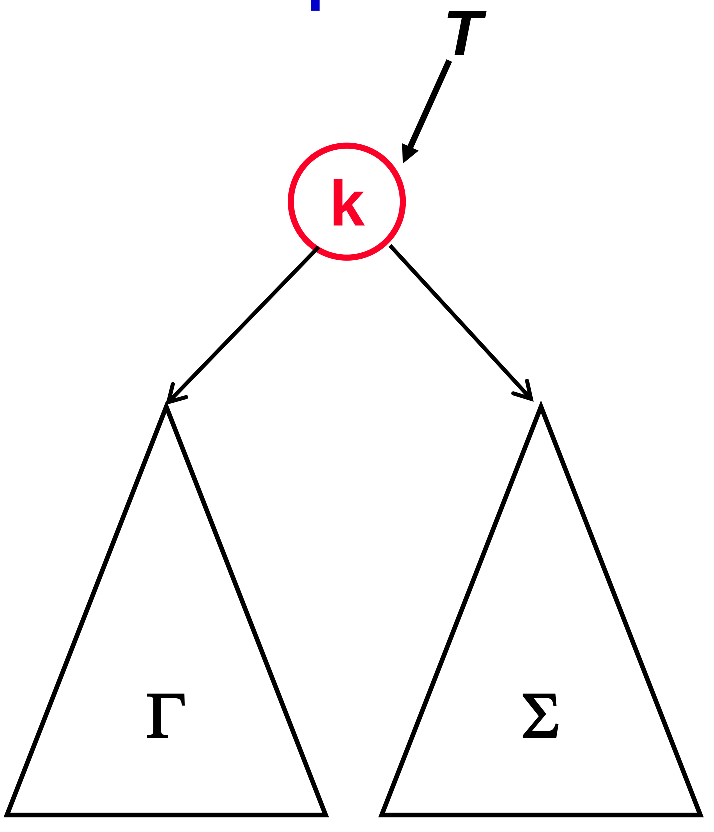
# *Cancellazione ricorsiva*



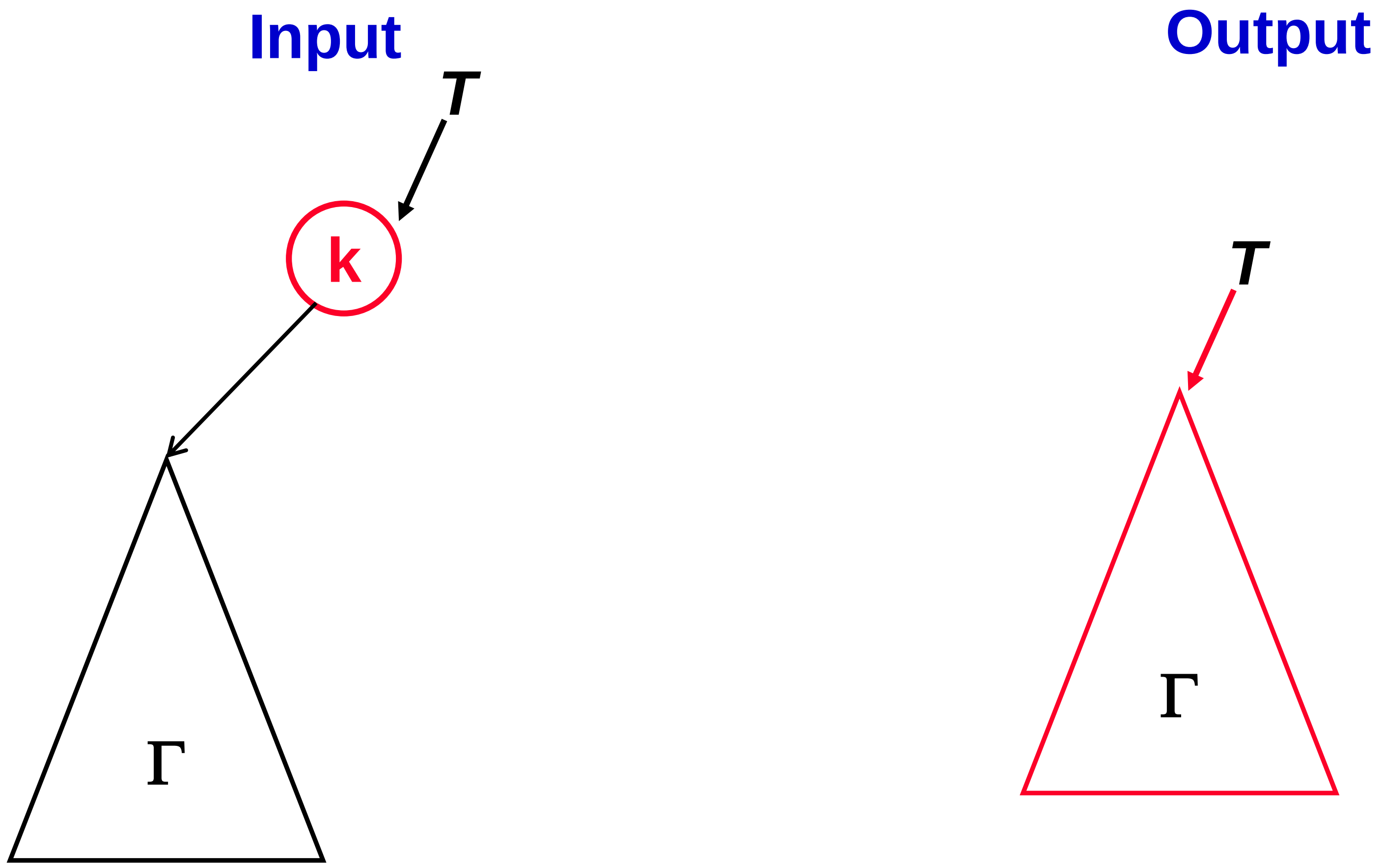

**Γ’ = cancella(Γ,k)**

# *Cancellazione ricorsiva*

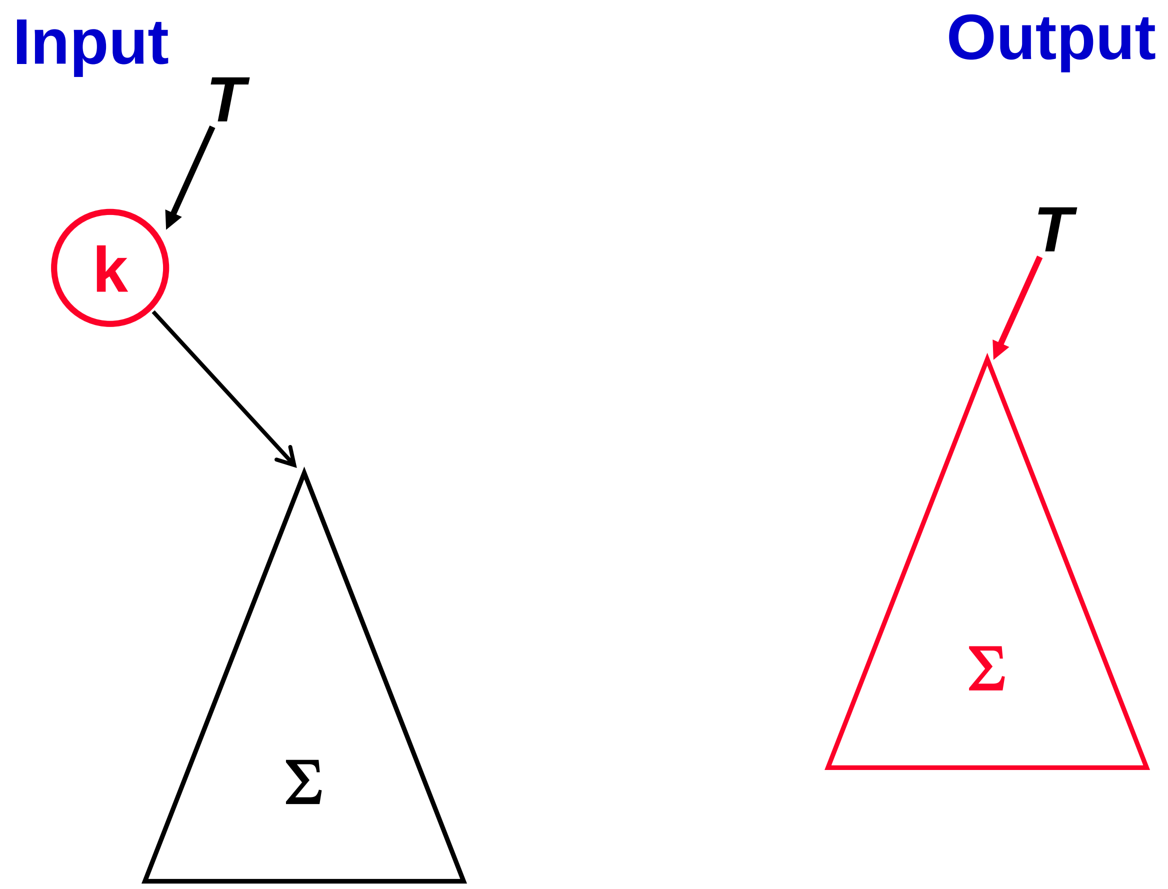
**Input**



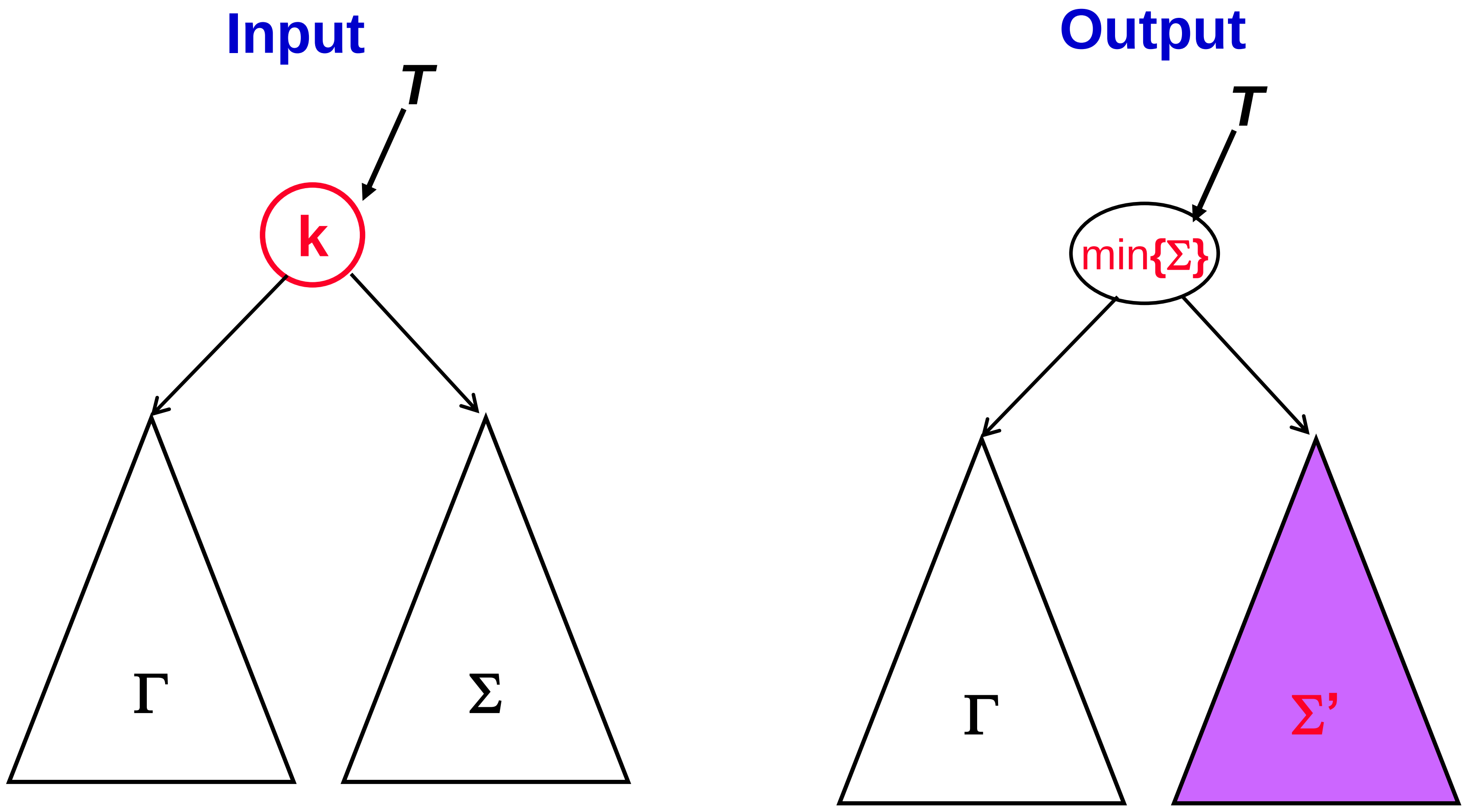
# *Cancellazione ricorsiva (caso I)*



# *Cancellazione ricorsiva (caso II)*



# *Cancellazione ricorsiva (caso III)*




**Σ' = stacca-minimo(Σ)**

# ARB: Cancellazione ricorsiva

key | sx | dx

```
ABR-Cancella-ric(k,T)
   IF T ≠ NIL THEN
     IF k < T->key THEN
          T->sx = ARB-Cancella-ric(k,T->sx)
     ELSE IF k > T->key THEN
          T->sx = ARB-Cancella-ric(k,T->dx)
     ELSE /* k = T->key */
          T = ABR-Cancella-Root(T)
   return T
```

**Ricerca successore Caso III**

```
ABR-Cancella-Root(T)
   IF T ≠ NIL THEN
      IF T->sx ≠ NIL && T->dx ≠ NILL THEN
           tmp = Stacca-Min(T,T->dx)
           "Copia tmp->key in T->key"
      ELSE
          tmp = T
          IF T->dx ≠ NIL THEN T = T->dx
                         ELSE T = T->sx
      dealloca tmp
   return T
```

**Casi I e II**

# *ARB: Cancellazione ricorsiva*

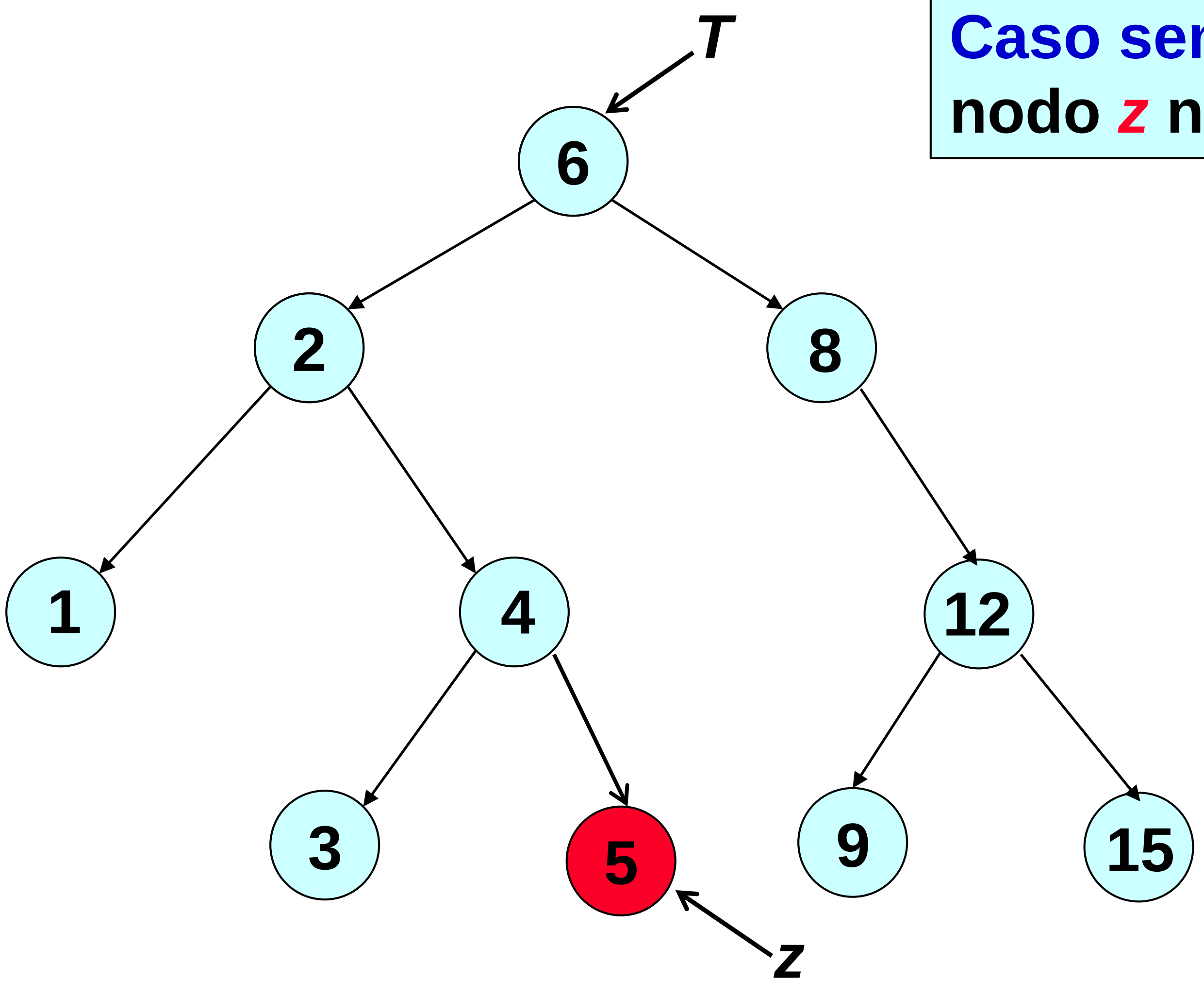
**Il parametro P serve per ricordarsi il padre di T durante la discesa**

```
Stacca-min(T,P)
   IF T ≠ NIL THEN
      IF T->sx ≠ NIL THEN
           return Stacca-min(T->sx,T)
      ELSE /* successore trovato */
         IF T = P->sx THEN
              P->sx = T->dx
         ELSE /* min è il primo nodo passato */
              P->dx = T->dx
   return T
```

***NOTA**. L'algoritmo stacca il nodo minimo dell'albero T e ne ritorna il puntatore. Può anche ritornare NIL in caso non esista un minimo (T è vuoto). Il valore di ritorno dovrebbe essere quindi verificato dal chiamante prima dell'uso.*

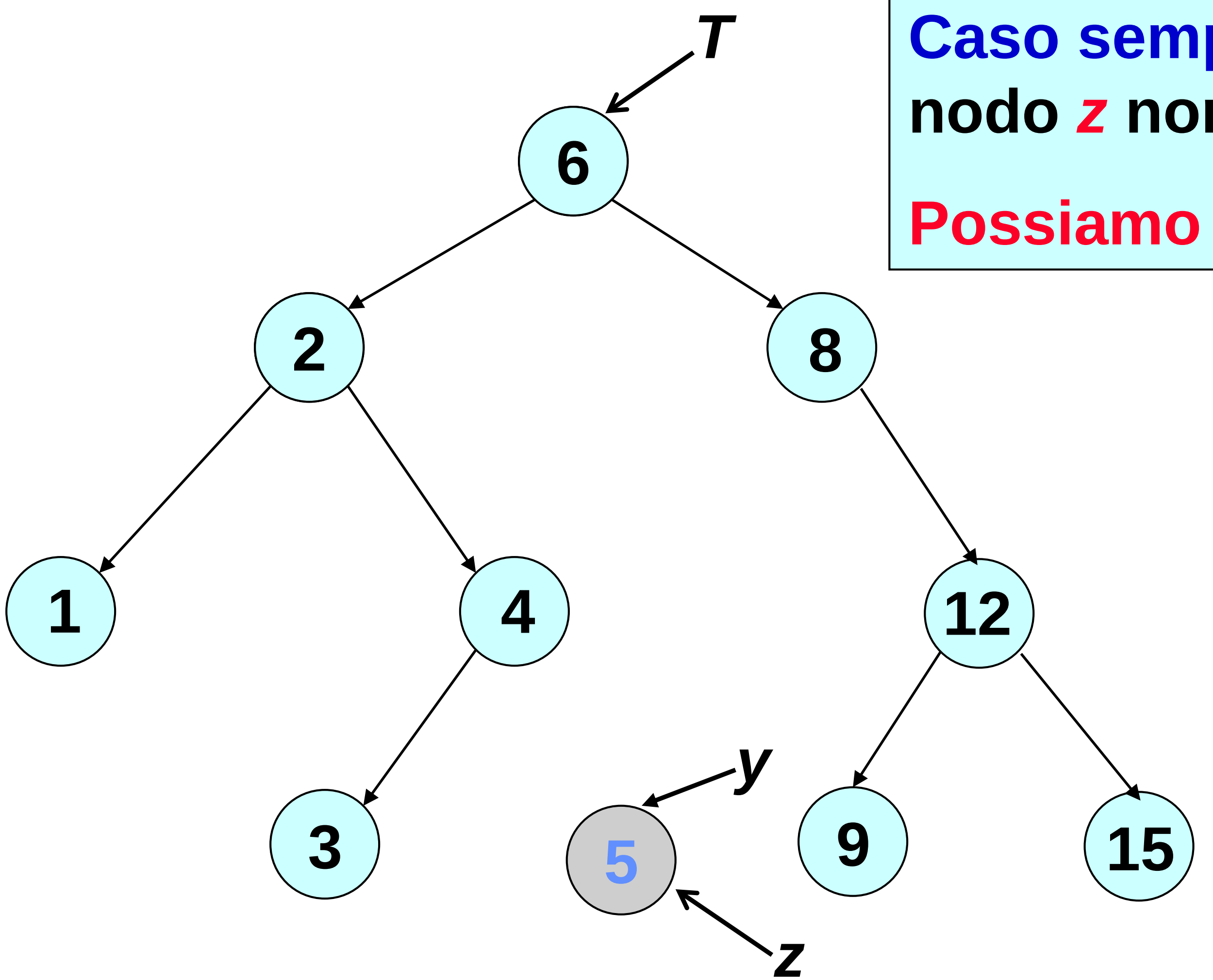
***Nel caso della cancellazione ricorsiva però siamo sicuri che il minimo esiste sempre e quindi non è necessario eseguire alcun controllo!***

# *ARB: Cancellazione di un nodo (caso I)*




**Caso semplice: il nodo *z* non ha figli**

# *ARB: Cancellazione di un nodo (caso I)*

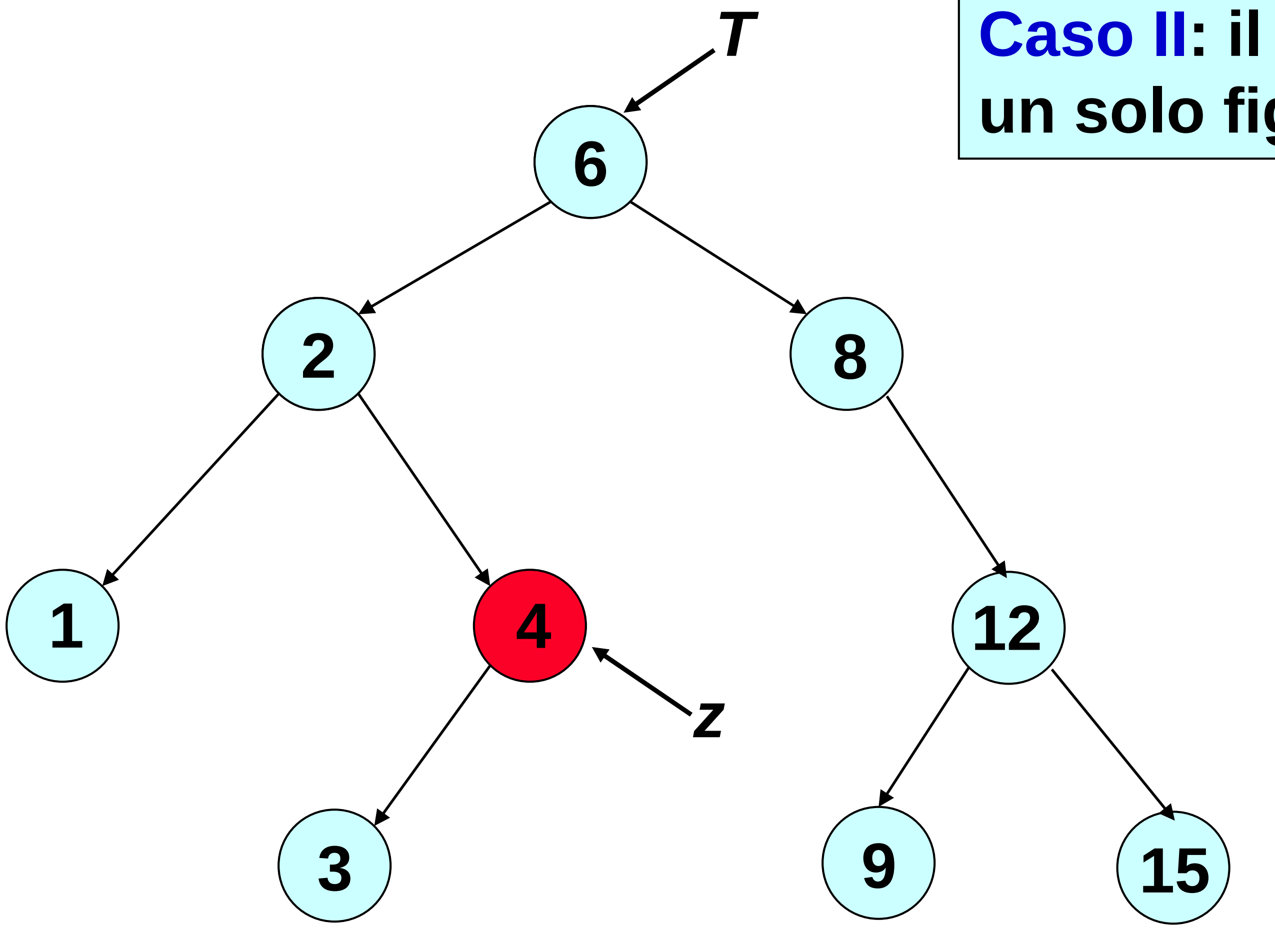



**Caso semplice: il nodo *z* non ha figli**

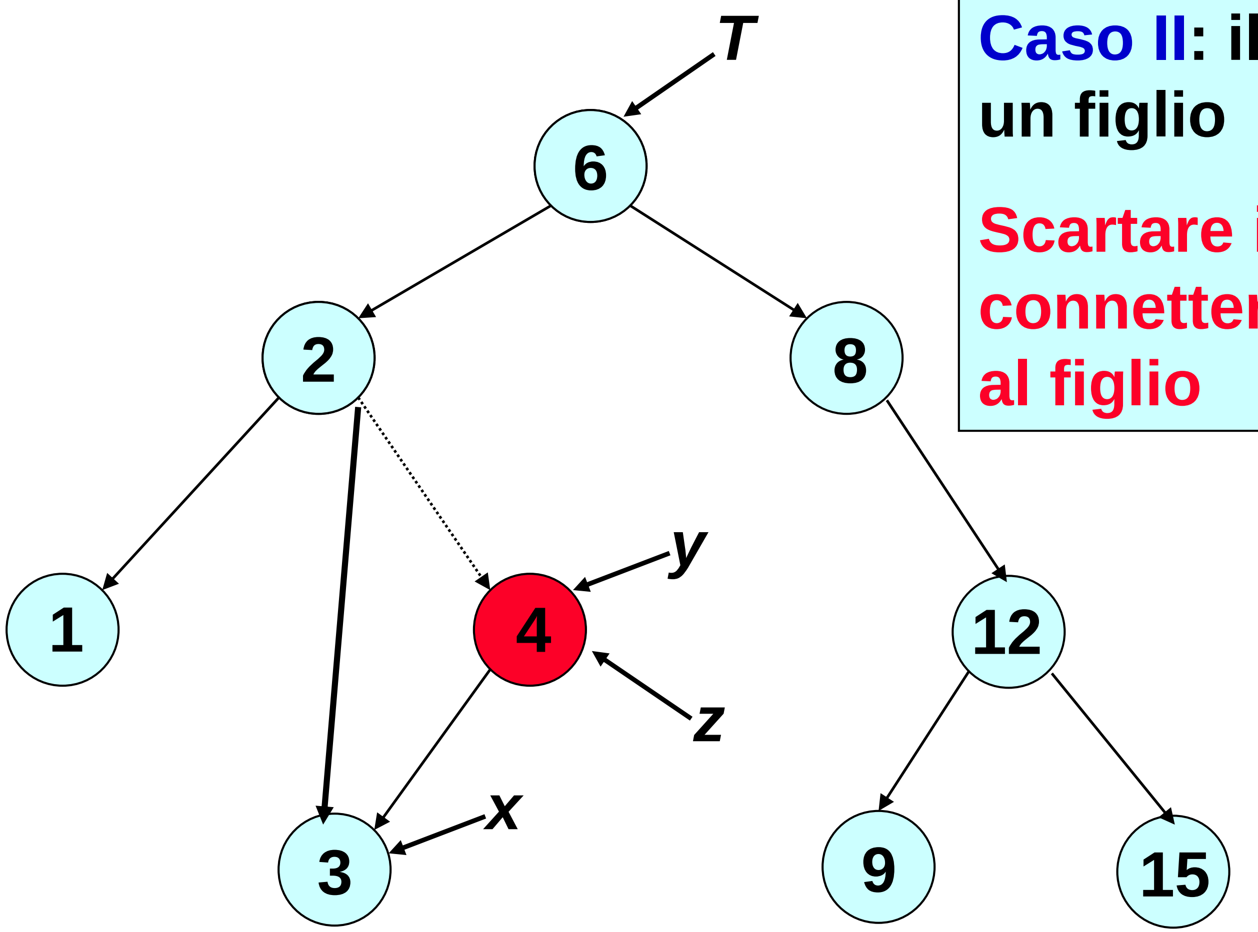
**Possiamo eliminarlo**

# *ARB: Cancellazione di un nodo (caso II)*

**Caso II: il nodo ha un solo figlio**



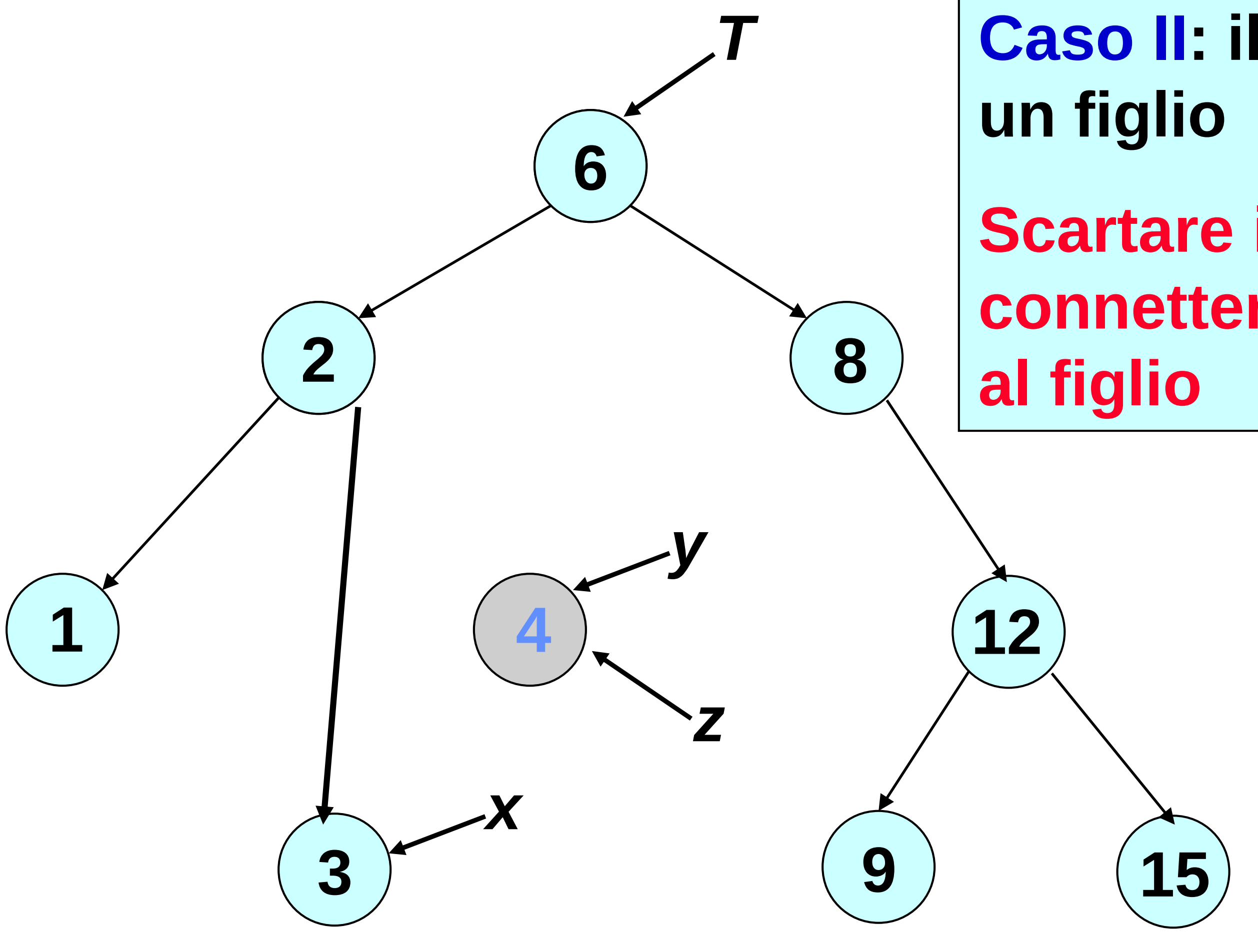
# *ARB: Cancellazione di un nodo (caso II)*




**Caso II: il nodo ha un figlio**

**Scartare il nodo e connettere il padre al figlio**

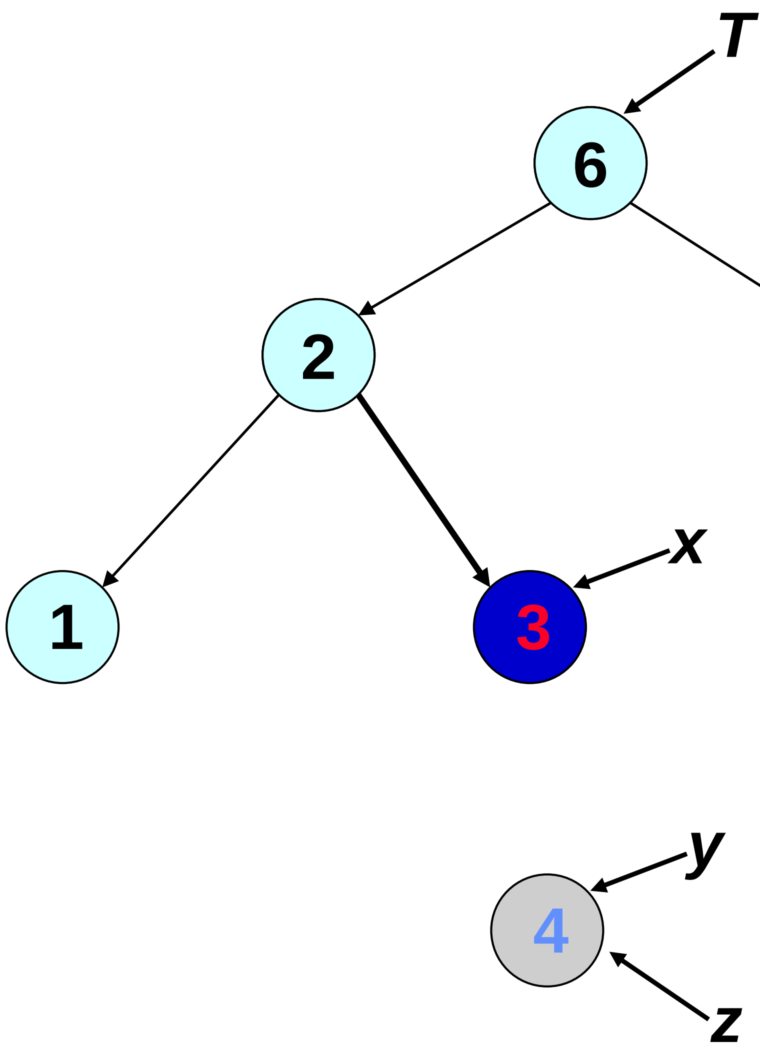
# *ARB: Cancellazione di un nodo (caso II)*




**Caso II: il nodo ha un figlio**

**Scartare il nodo e connettere il padre al figlio**

# *ARB: Cancellazione di un nodo (caso II)*

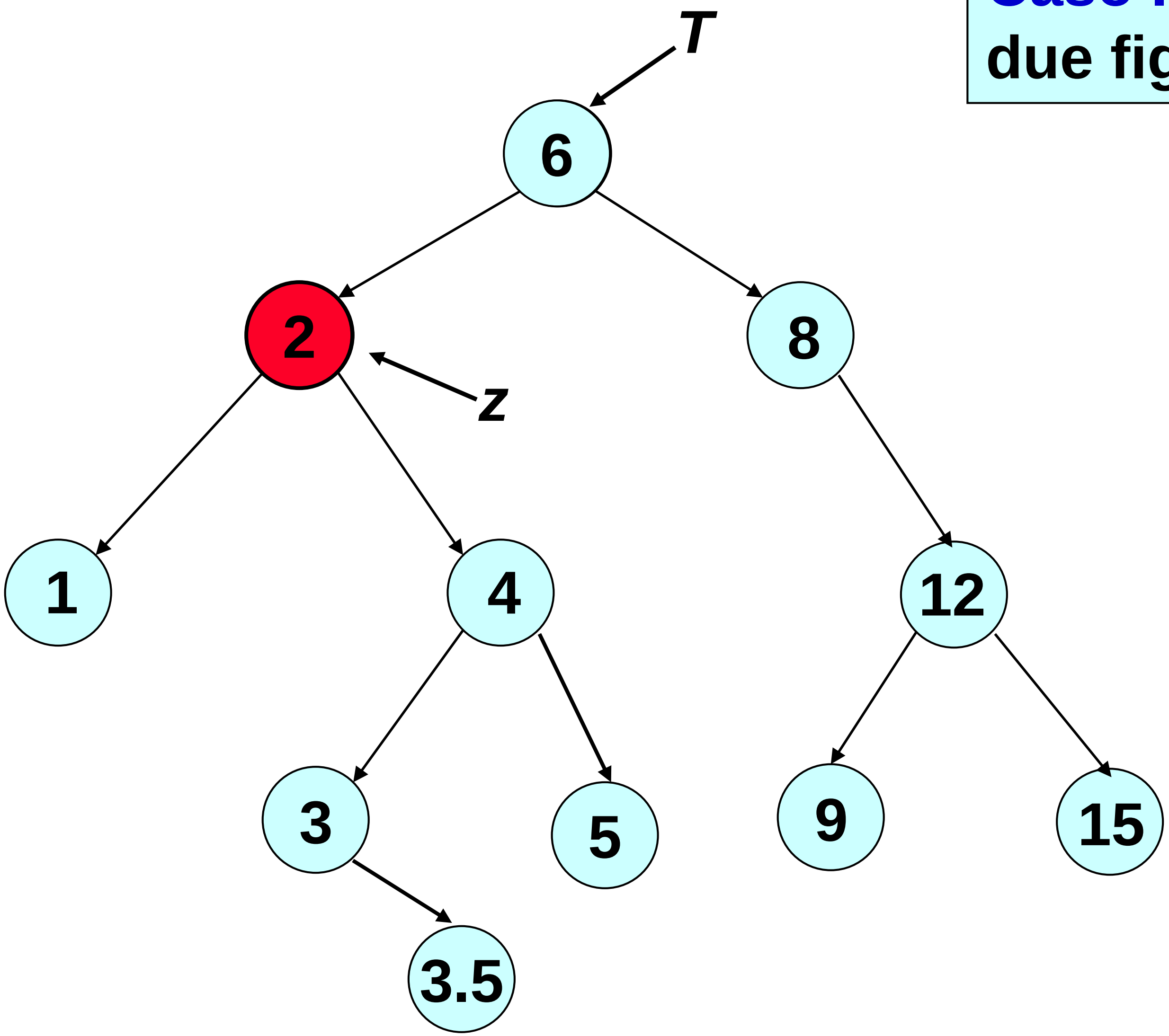



**Caso II: il nodo ha un figlio**

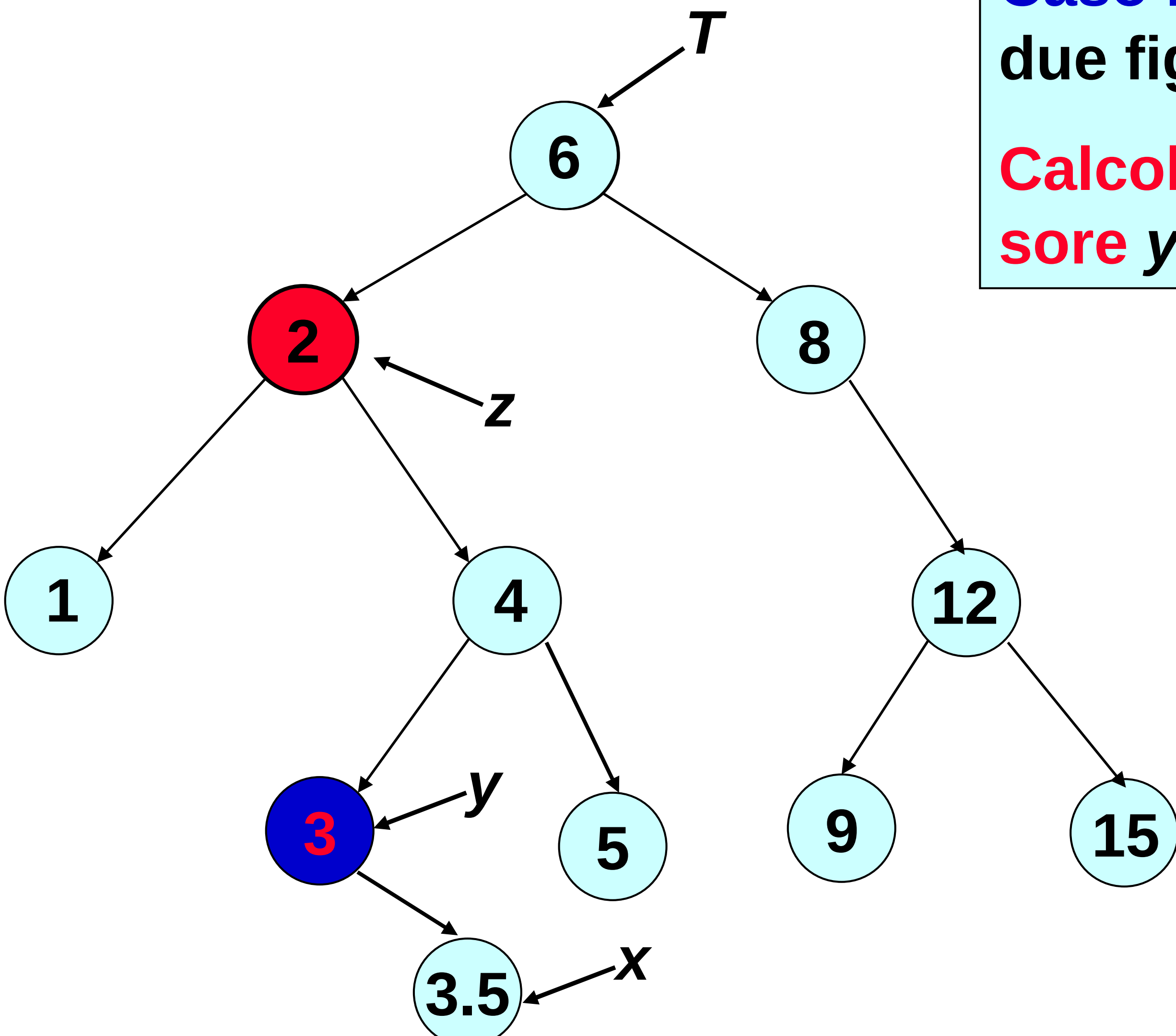
**Scartare il nodo e connettere il padre al figlio**

# *ARB: Cancellazione di un nodo (caso III)*

**Caso III: il nodo ha due figli**



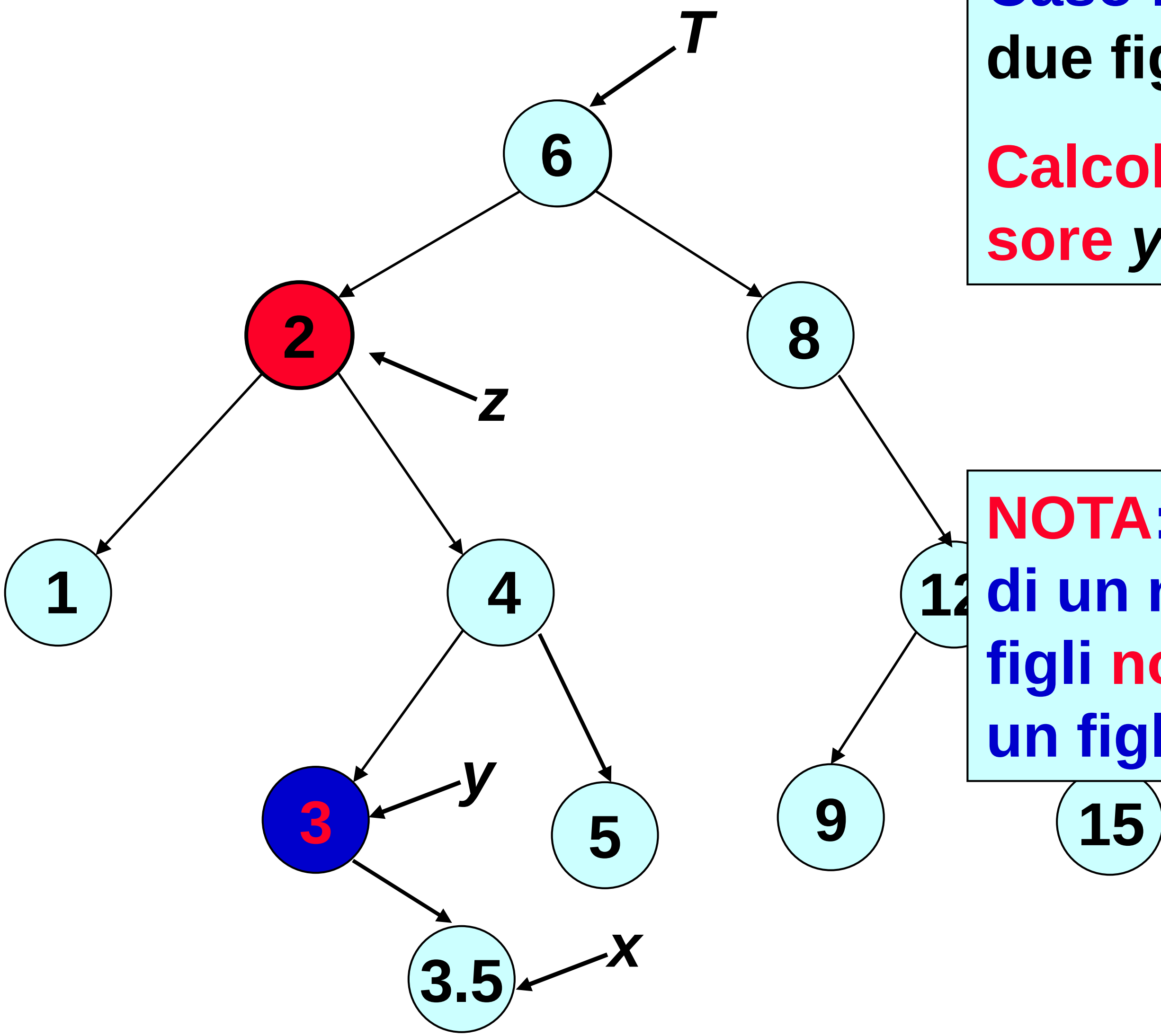
# *ARB: Cancellazione di un nodo (caso III)*




**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

# ARB: Cancellazione di un nodo (caso III)

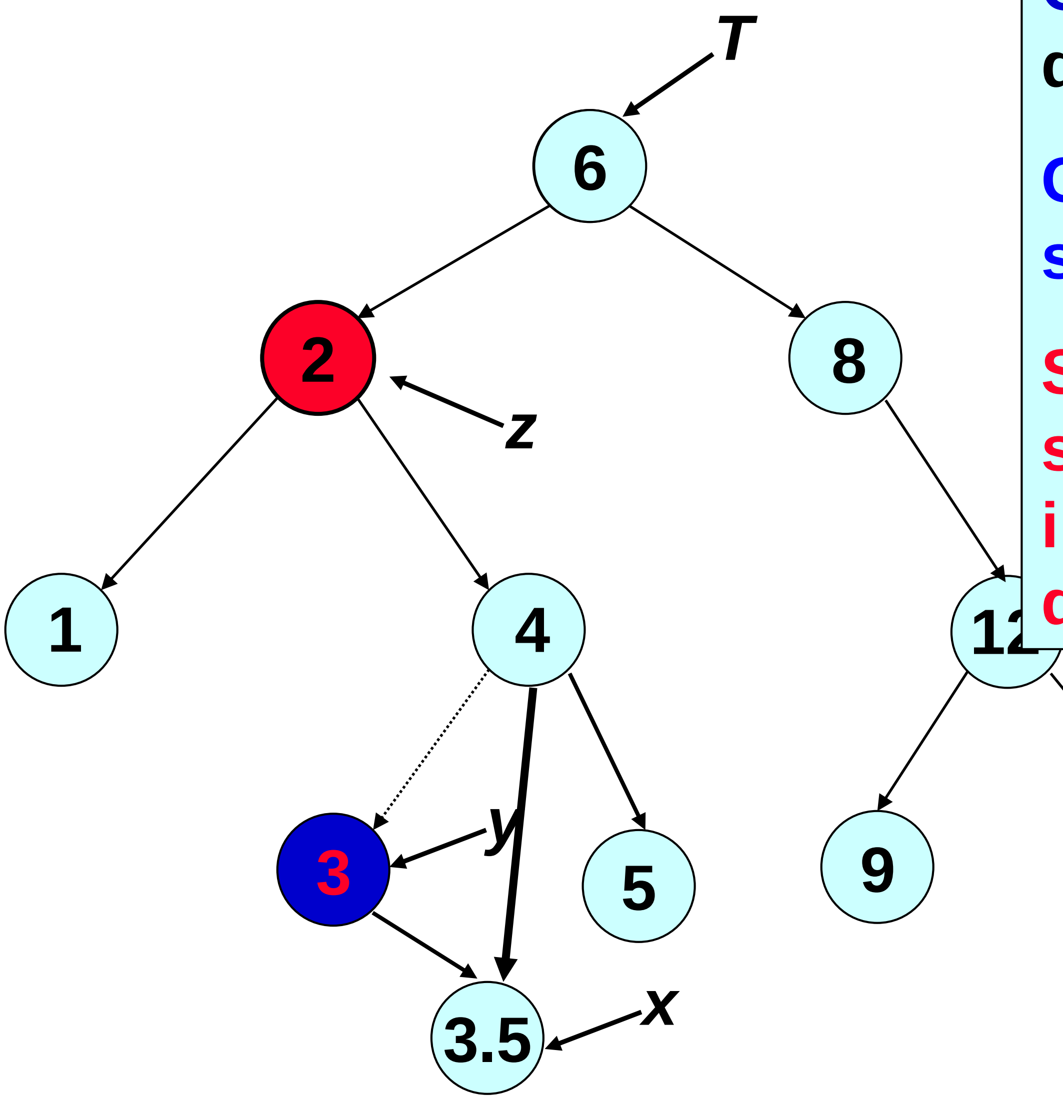



**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

**NOTA: Il successore di un nodo con due figli non può avere un figlio sinistro**

# *ARB: Cancellazione di un nodo (caso III)*

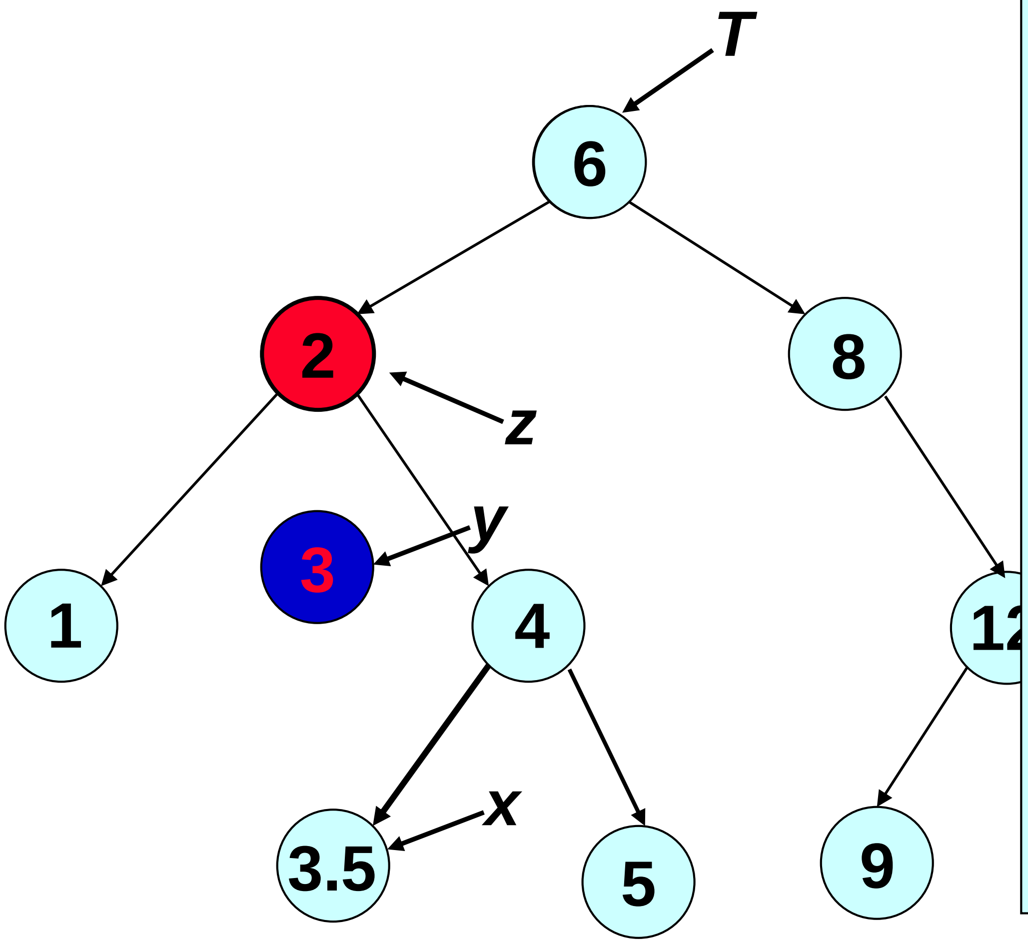



**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

**Staccare il successore *y* e connettere il padre al figlio destro**

# ARB: Cancellazione di un nodo (caso III)



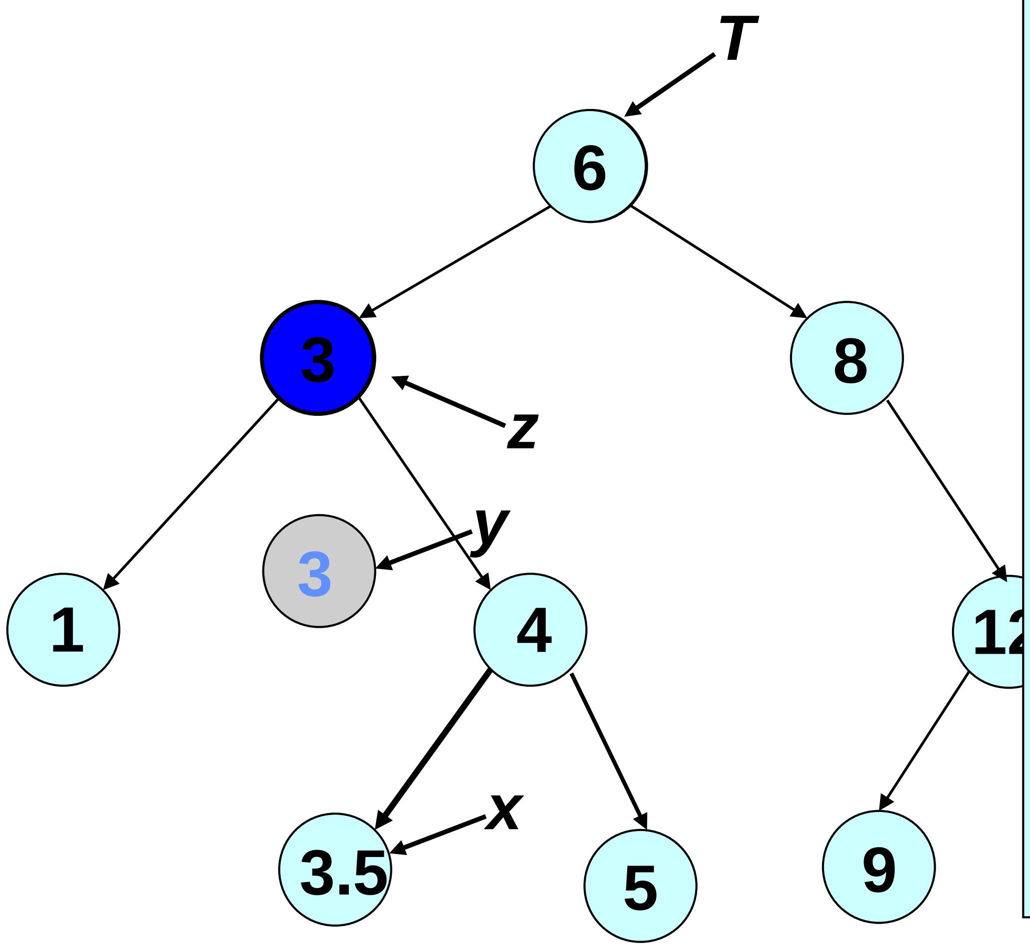

**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

**Staccare il successore *y* e connettere il padre al figlio destro**

**Copia il contenuto del successore nel nodo da cancellare**

# *ARB: Cancellazione di un nodo (caso III)*



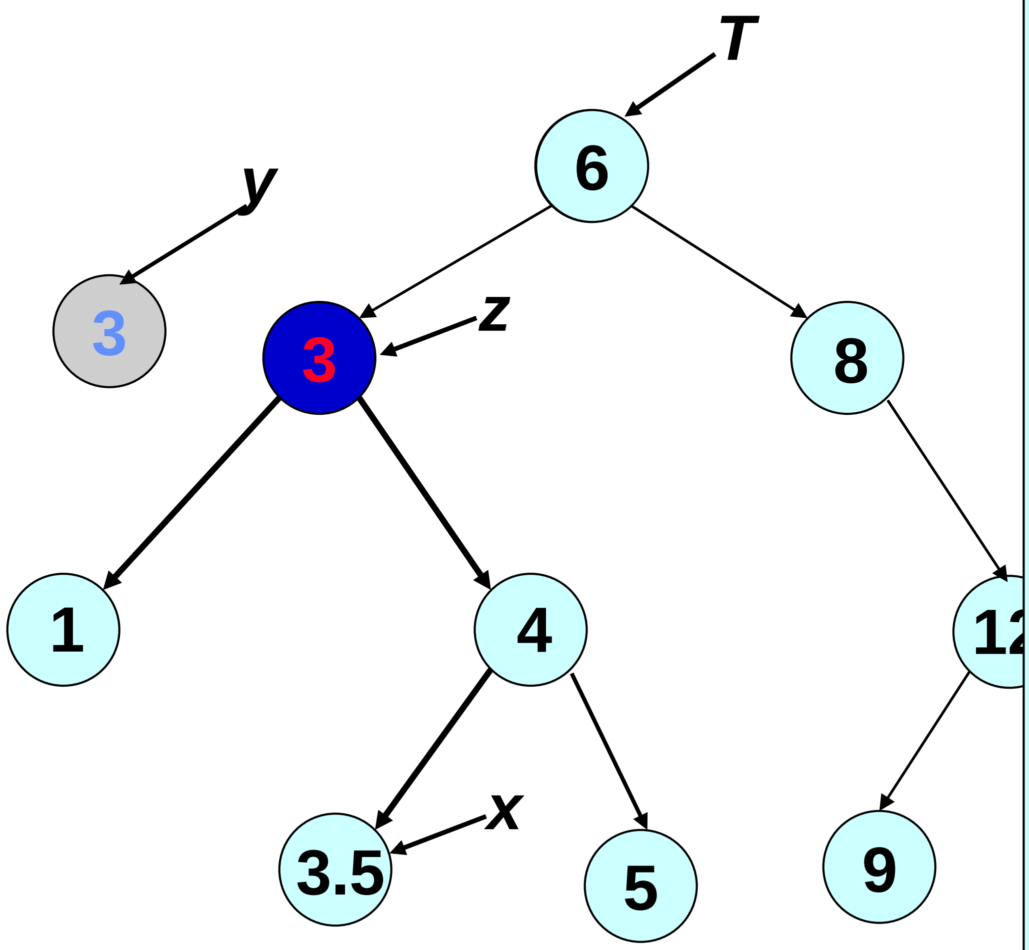

**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

**Staccare il successore *y* e connettere il padre al figlio destro**

**Copia il contenuto del successore nel nodo da cancellare**

# *ARB: Cancellazione di un nodo (caso III)*




**Caso III: il nodo ha due figli**

**Calcolare il successore *y***

**Staccare il successore *y* e connettere il padre al figlio destro**

**Copia il contenuto del successore *y* nel nodo da cancellare**

**Deallocare il nodo staccato *y***

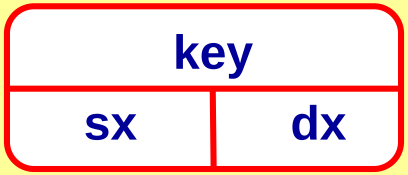
# *ARB: Cancellazione di un nodo*

- ***Caso I*: Il nodo non ha figli. Semplicemente si elimina.**

- ***Caso II*: Il nodo ha un solo figlio. Si *collega il padre del nodo al figlio* e si elimina il nodo.**

- ***Caso III*: Il nodo ha due figli.**
  - **si *cerca il suo successore* (che ha *un solo figlio destro*);**
  - **si *elimina il successore* (come in *Caso II*);**
  - **si *copiano* i campi *valore* del successore *nel nodo* da eliminare.**

```
ABR-Cancella-iter(T,k)
    P = NIL
    z = T
    WHILE (z ≠ NIL && z->key ≠ k) DO
          P = z
          IF (z->key > k) THEN z = z->sx
                          ELSE z = z->dx
    IF (z ≠ NIL) THEN /* k trovato */
       x = ABR-Cancella-Root(z)
       IF P = NIL THEN T = x /* si cancella la radice di T */
       ELSE IF z = P->sx THEN P->sx = x
                         ELSE P->dx = x
    return T
```




Ricerca successore
Caso III

```
ABR-Cancella-Root(T)
    IF T ≠ NIL THEN
       IF T->sx ≠ NIL && T->dx ≠ NILL THEN
            tmp = Stacca-Min(T,T->dx)
            "Copia tmp->key in T->key"
       ELSE
            tmp = T
            IF T->dx ≠ NIL THEN T = T->dx
                           ELSE T = T->sx
       dealloca tmp
    return T
```

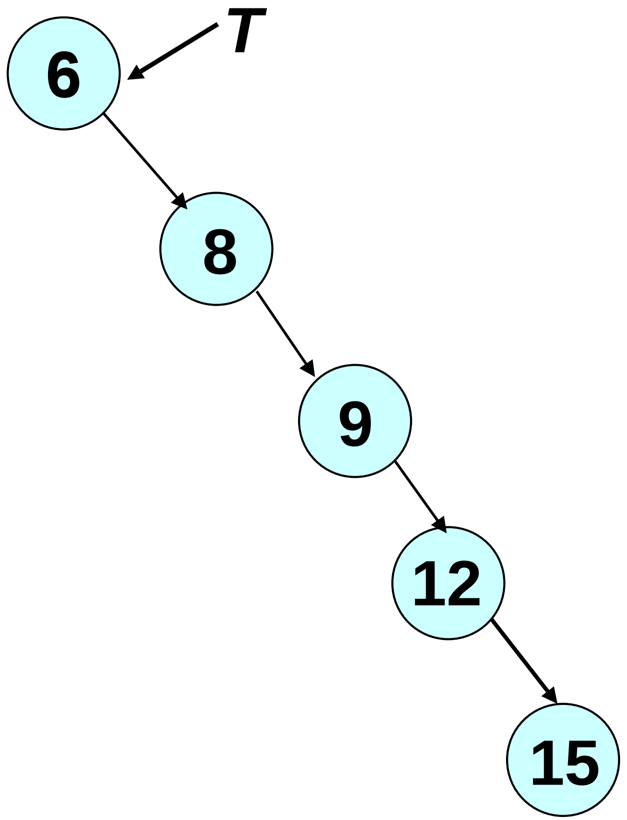
Casi I e II

# *ARB: costo di Inserimento e Cancellazione*

***Teorema**. Le operazioni di Inserimento e Cancel-lazione sull'insieme dinamico Albero Binario di Ricerca possono essere ese-guite in tempo $O(h)$ dove $h$ è l'altezza dell'albero*

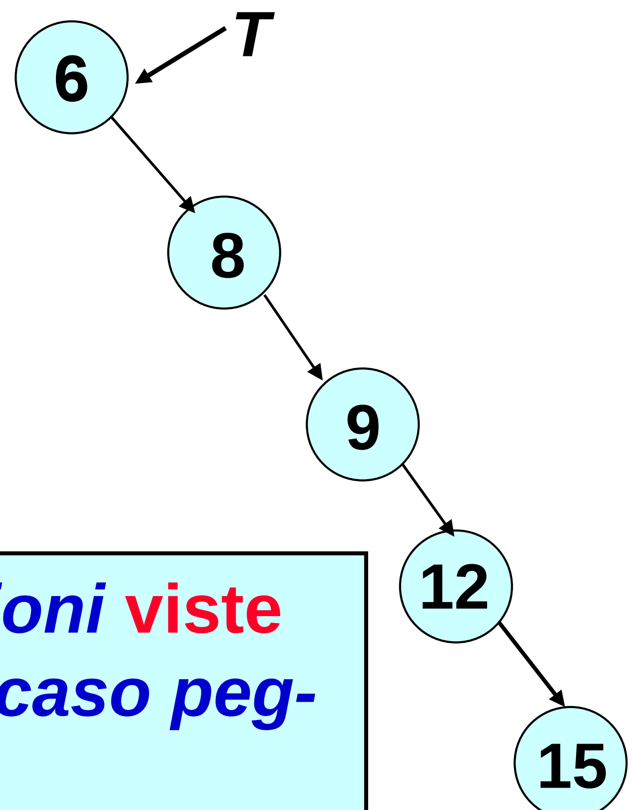
# *Costo delle operazioni su ABR*

***L'algortimo di inserimento NON garantisce che l'albero risultante sia bilaciato. Nel caso peggiore l'altezza h può essere pari ad N (numero dei nodi)***



# *Costo delle operazioni su ABR*

***L'algortimo di inserimento NON garantisce che l'albero risultante sia bilaciato. Nel caso peggiore l'altezza h può essere pari ad N (numero dei nodi)***




**Quindi *tutte le operazioni* viste hanno *costo* O(*N*) nel *caso peggiore***

# *Costo medio delle operazioni su ABR*

**Dobbiamo calcolare la *lunghezza media* $a(n)$ del *percorso di ricerca*.**

- **Assumiamo che le chiavi arrivino in ordine casuale (e che tutte abbiano *uguale probabilità* di presentarsi)**
- **La probabilità che la chiave *i* sia la radice è allora 1/*n***

$$a(n) = \frac{1}{n} \sum_{j=1}^{n} p_j$$

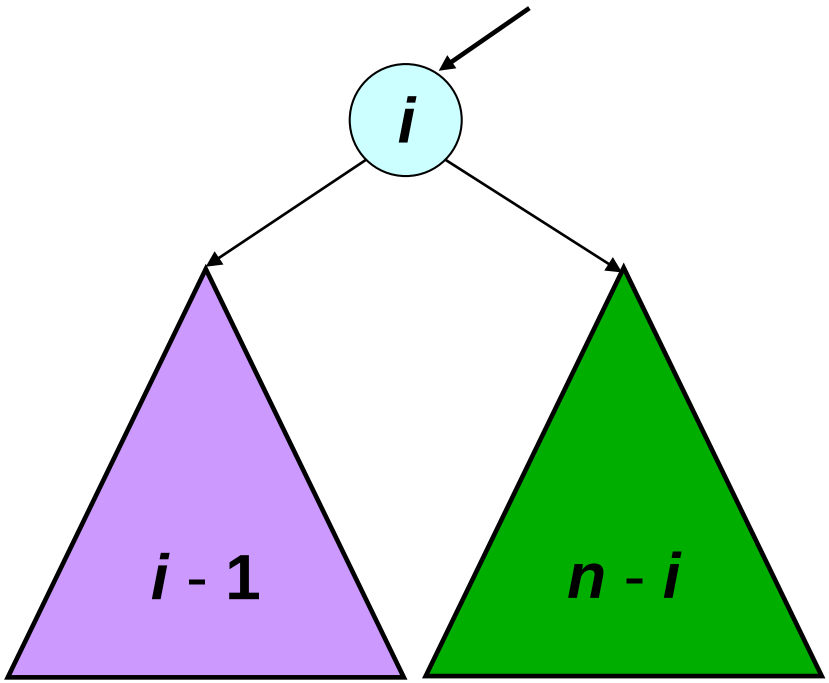
**$p_j$ è la lunghezza media del percorso al nodo *j***

# Costo delle operazioni su ABR

Se *i* è la radice, allora

- il sottoalbero sinistro avrà *i* - 1 nodi e
- il sottoalbero destro avrà *n* - *i* nodi

$$a(n) = \frac{1}{n} \sum_{j=1}^{n} p_j$$



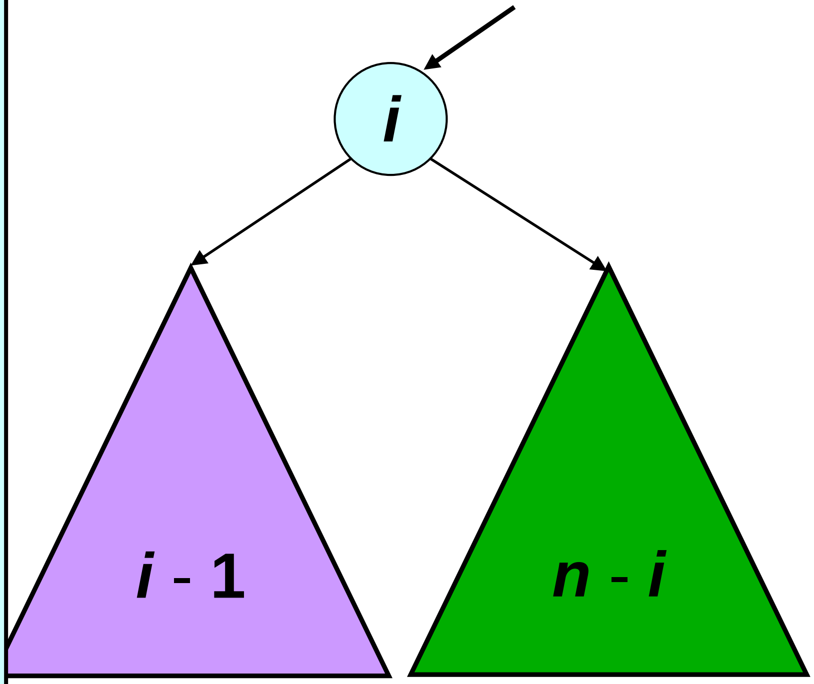
# Costo delle operazioni su ABR

Se *i* è la radice, allora

- il sottoalbero sinistro avrà *i* - 1 nodi e
- il sottoalbero destro avrà *n* - *i* nodi

$$a(n) = \frac{1}{n} \sum_{j=1}^{n} p_j$$

- gli *i* - 1 nodi a sinistra hanno lunghezza media del percorso $a(i-1)+1$
- la radice ha lunghezza del percorso pari ad 1
- gli *n* - *i* nodi a sinistra hanno lunghezza media del percorso $a(n-i)+1$



# Costo delle operazioni su ABR

$a^i(n)$ **sia la lunghezza media del percorso di ricerca con** $n$ **chiavi quando la radice è la chiave** $i$

$$a^i(n) = [a(i-1)+1]\frac{i-1}{n} + 1\frac{1}{n} + [a(n-i)+1]\frac{n-i}{n}$$

$a(i-1)$ **è la lunghezza media del percorso di ricerca con** $i-1$ **chiavi**

$a(n-i)$ **è la lunghezza media del percorso di ricerca con** $n-i$ **chiavi**

# Costo delle operazioni su ABR

$a^i(n)$ **sia la lunghezza media del percorso di ricerca con** $n$ **chiavi quando la radice è la chiave** $i$

$$a^i(n) = [a(i-1)+1]\frac{i-1}{n} + 1\frac{1}{n} + [a(n-i)+1]\frac{n-i}{n}$$

***allora***

$$a(n) = \frac{1}{n}\sum_{i=1}^{n} a^i(n)$$

$a(n)$ **è la media degli** $a^i(n)$**, dove ciascun** $a^i(n)$ **ha probabilità 1/*n*, cioè la probabilità che proprio la chiave** $i$ **sia la radice dell'albero.**

# Costo delle operazioni su ABR

$$a^i(n) = [a(i-1)+1]\frac{i-1}{n} + 1\frac{1}{n} + [a(n-i)+1]\frac{n-i}{n}$$

*allora*

$$a(n) = \frac{1}{n}\sum_{i=1}^{n} a^i(n) =$$

$$= \frac{1}{n}\sum_{i=1}^{n}[a(i-1)+1]\frac{i-1}{n} + 1\frac{1}{n} + [a(n-i)+1]\frac{n-i}{n}$$

***a(n)*** **è la media degli** ***$a^i(n)$*****, dove ciascun** ***$a^i(n)$*** **ha probabilità 1/*n***

# Costo delle operazioni su ABR

$$\boxed{a(n) = \frac{1}{n}\sum_{i=1}^{n}[a(i-1)+1]\frac{i-1}{n} + 1\frac{1}{n} + [a(n-i)+1]\frac{n-i}{n}}$$

$$= 1 + \frac{1}{n^2}\sum_{i=1}^{n}\left[a(i-1)\cdot(i-1) + a(n-i)\cdot(n-i)\right]$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n}\left[a(i-1)\cdot(i-1)\right]$$

$$\boxed{= 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i a(i)}$$

# Costo delle operazioni su ABR

$$a(n) = 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} i \cdot a(i)$$

$$= 1 + \frac{2}{n^2}(n-1) \cdot a(n-1) + \frac{2}{n^2} \sum_{i=0}^{n-2} i \cdot a(i)$$

# Costo delle operazioni su ABR

$$a(n) = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i \cdot a(i)$$

$$= 1 + \frac{2}{n^2}(n-1)\cdot a(n-1) + \frac{2}{n^2}\sum_{i=0}^{n-2} i \cdot a(i)$$

$$a(n-1) = 1 + \frac{2}{(n-1)^2}\sum_{i=0}^{n-2} i \cdot a(i)$$

# Costo delle operazioni su ABR

$$a(n) = 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} i \cdot a(i)$$

$$= 1 + \frac{2}{n^2}(n-1) \cdot a(n-1) + \frac{2}{n^2} \sum_{i=0}^{n-2} i \cdot a(i)$$

$$\frac{2}{n^2} \sum_{i=0}^{n-2} i \cdot a(i) = \frac{(n-1)^2}{n^2}\big(a(n-1) - 1\big)$$

$$a(n-1) = 1 + \frac{2}{(n-1)^2} \sum_{i=0}^{n-2} i \cdot a(i)$$

# Costo delle operazioni su ABR

$$a(n) = 1 + \frac{2}{n^2}(n-1)\cdot a(n-1) + \frac{2}{n^2}\sum_{i=0}^{n-2} i \cdot a(i)$$

$$\frac{2}{n^2}\sum_{i=0}^{n-2} i \cdot a(i) = \frac{(n-1)^2}{n^2}\big(a(n-1)-1\big)$$

$$a(n) = \frac{1}{n^2}\left[(n^2-1)\cdot a(n-1) + 2n - 1\right]$$

## *Costo delle operazioni su ABR*

$$a(n) = \frac{1}{n^2}\left[(n^2 - 1)\cdot a(n-1) + 2n - 1\right]$$

**Dimostrare per induzione**

$$a(n) = 2\frac{n+1}{n}H(n) - 3$$

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

**Funzione armonica**

# Costo delle operazioni su ABR

$$a(n) = 2\frac{n+1}{n}H(n) - 3$$

$$\boxed{a(n) = 2(\ln n + \gamma) - 3 = 2\ln n - c}$$

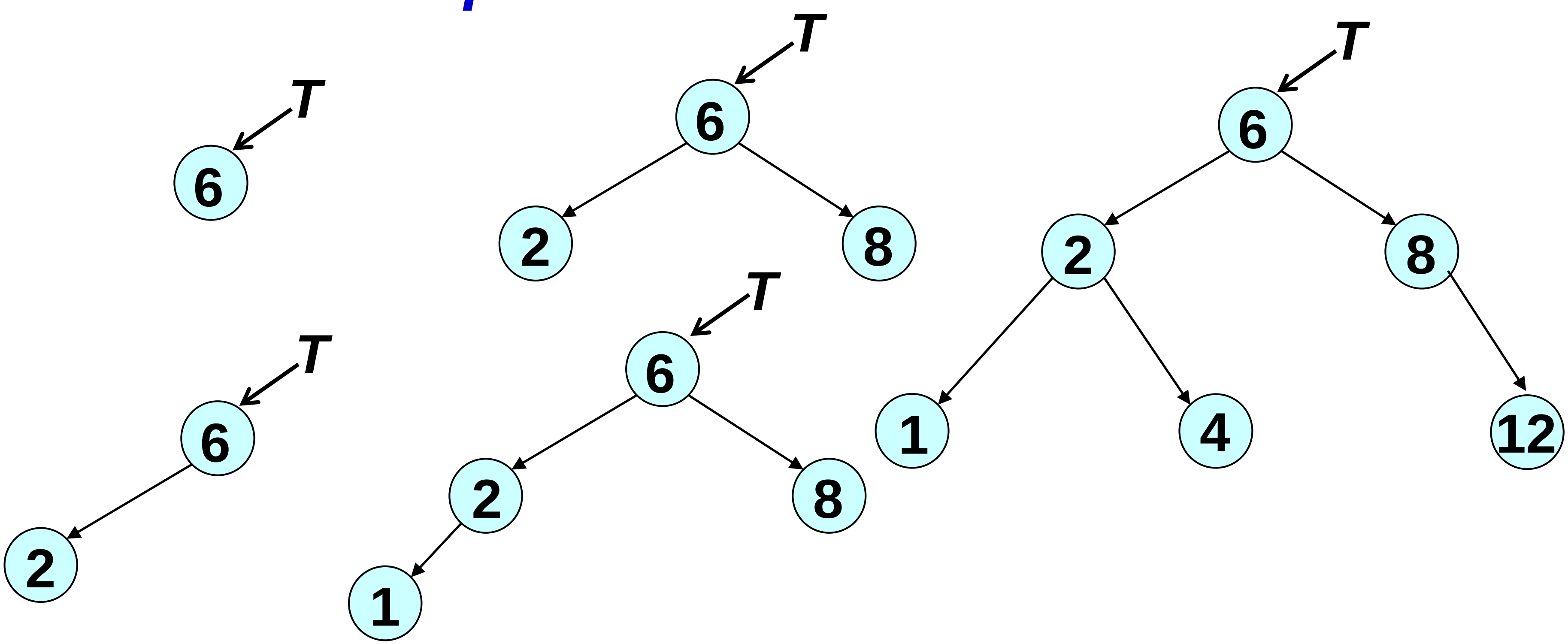
Formula di Eulero

$$H(n) = \gamma + \ln n + \frac{1}{2n} - \frac{1}{12n^2} + \ldots$$

**dove** $\gamma \approx 0.577$

# *Alberi perfettamente bilanciati*

***Definizione:*** **Un albero binario si dice *Perfetta-mente Bilanciato* se, per ogni nodo *i*, il *numero dei nodi* nel suo *sottoalbero sinistro* e il *numero dei nodi* del suo *sottoalbero destro* *differiscono al più* di 1**



# *Alberi perfettamente bilanciati*

***Definizione:*** **Un albero binario si dice *<u>Perfettamente Bilanciato</u>* se, per ogni nodo *i*, il *numero dei nodi* nel suo *sottoalbero sinistro* e il *numero dei nodi* del suo *sottoalbero destro* *differiscono al più* di 1**

**La *lunghezza media $a'(n)$ del percorso* in un *albero perfettamente bilanciato (APB)* con *n* nodi è approssimativamente**

$$\boxed{a'(n) = \log n - 1}$$

## *Confronto tra ABR e APB*

**Il *rapporto* tra la *lunghezza media* $a(n)$ *del percorso* in un *albero di ricerca* e la *lunghezza media* $a'(n)$ nell'*albero perfettamente bilanciato* è (per $n$ sufficientemente grande) è approssimativamente**

$$\boxed{\frac{a(n)}{a'(n)} = \frac{2\ln n - c}{\log n - 1} \cong \frac{2\ln n}{\log n} = 2\ln 2 \cong 1{,}386}$$

**(trascurando i termini costanti)**

## *Confronto tra ABR e APB*

**Ciò significa che, se anche *bilanciassimo* perfettamente l'albero *dopo ogni inserimento* il *guadagno sul percorso medio* che otterremmo *NON supererebbe* il 39%.**

$$\frac{a_n}{a'_n} = \frac{2\ln n - c}{\log n - 1} = \frac{2\ln n}{\log n} = 2\ln 2 \simeq 1.386$$

***Sconsigliabile* nella maggior parte dei casi, *a meno che* il *numero dei nodi* e il rapporto tra ricerche e inserimenti *siano molto grandi*.**